Software Defined Networking

Applicability and Service Possibilities

Caba, Cosmin Marius

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

PhD Thesis

Software Defined Networking

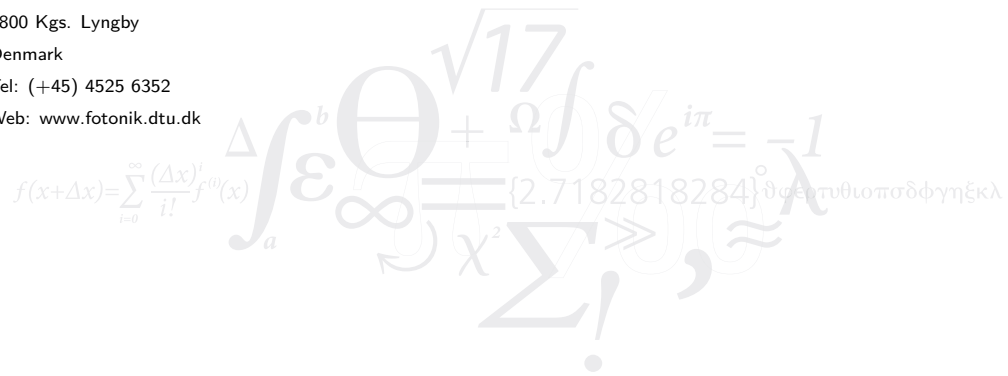
Applicability and Service Possibilities

Cosmin Marius Caba

Technical University of Denmark
Kgs. Lyngby, Denmark 2016

Cover image taken from: http://cloudtimes.org/2014/09/29/software-defined-networking-market-soars-as-enterprises-adopt-network-virtualization/.

Supervisors:
José Soler
Lars Dittmann
Technical University of Denmark
Department of Photonics Engineering
Ørsteds Plads 343
2800 Kgs. Lyngby
Denmark
Tel: (+45) 4525 6352
Web: www.fotonik.dtu.dk



Abstract

Network Service Providers (NSP) often choose to overprovision their networks instead of deploying proper Quality of Services (QoS) mechanisms that allow for traffic differentiation and predictable quality. This tendency of overprovisioning is not sustainable for the simple reason that network resources are limited. Hence, to counteract this trend, current QoS mechanisms must become simpler to deploy and operate, in order to motivate NSPs to employ QoS techniques instead of overprovisioning.

Software Defined Networking (SDN) represents a paradigm shift in the way telecommunication and data networks are designed and managed. This thesis argues that SDN can greatly simplify QoS provisioning in communication networks, and even improve QoS in various ways. To this end, the impact of SDN on QoS is assessed from both a network performance perspective (e.g. bandwidth, delay), and also from a more generic perspective (e.g. service provisioning speed, resources availability). As a result, new mechanisms for providing QoS are proposed, solutions for SDN-specific QoS challenges are designed and tested, and new network management concepts are prototyped, all aiming to improve QoS for network services, from this extended point of view.

Specifically, the challenge of SDN based QoS provisioning is addressed by considering every layer of the SDN architecture. In chapter 2, a short introduction to SDN is given, following that a complete architecture for QoS aware service provisioning in SDN is introduced in chapter 3. The following three chapters (4, 5 and 6) focus on each logical plane of the SDN architecture and identify the major challenges with respect to QoS, in relation to that specific plane (i.e. data, control and management). Further, each chapter proposes solutions to address the identified challenges, and demonstrates these solutions by testing them in various network scenarios. The last chapter of the thesis concentrates on applying SDN to improve QoS and increase the network utilization in a novel data center environment. This environment comprises a hybrid packet-circuit architecture, on top of which intelligent algorithms are applied in order to selectively offload traffic from the capacity constrained packet based network onto optical circuits.

Overall, the research work presented in this thesis identifies and addresses the critical aspects of SDN based QoS provisioning. Moreover, several tests and demon-

strations have been performed by using virtualization techniques. These tests aim to support the proposed ideas, and also to create a better picture of practical SDN deployments and the difficulties that arise in such virtualized environments.

Resumé

Frem for at implementere Quality of Service (QoS) metoder med henblik på at differentiere datatrafik og dermed tilbyde en forudsigelig kvalitet, så vælger typiske internetleverandører ofte at overdimensionere deres netværk. Dette er dog på sigt ikke en holdbar løsning, da netværksressourcerne er begrænsede. For at imødekomme denne tendens, er det nødvendigt at de nuværende QoS mekanismer forenkles i forhold til implementation og drift for således at motivere leverandørerne til at implementere QoS teknikker frem for simpel overdimensionering.

Måden tele- og datanetværk designes og styres står overfor et paradigmeskift repræsenteret af teknologien Software Defined Networking (SDN). Denne afhandling hævder at SDN kan muliggøre en betydelig forenkling af kvalitetsstyring af kommunikationsnetværk og endda forbedre kvaliteten på en række områder. Det er således undersøgt hvilken indflydelse SDN har på QoS set både ud fra et ydeevneaspekt (f.eks. båndbredde, forsinkelse) og fra nogle mere generelle aspekter (f.eks. hastighed af tjenesteprovisionering, ressourcetilgængelighed). Som resultat af dette er foreslået nye metoder for QoS ligesom der er udviklet og valideret SDN baserede QoS- løsninger. Baseret på dette er udviklet nye netværksmanagementkoncepter med det formål at forbedre QoS for netværkstjenesterne.

Udfordringerne ved SDN baseret QoS-provisionering er helt specifikt adresseret ved at undersøge de enkelte lag i SDN-arkitekturen. Kapitel 2 giver en introduktion til SDN, hvilket efterfølges i kapitel 3 af en komplet arkitektur for SDN-baseret QoS orienteret tjenesteprovisionering. De følgende tre kapitler (4, 5 and 6) stiller skarpt på henholdsvis data-, kontrol- og managementplanet i SDN arkitekturen og identificerer de primære QoS-relevant udfordringer for hvert plan. Desuden er i hvert kapitel foreslået og testet løsninger på disse udfordringer i diverse netværksscenarier. Afhandlingens sidste kapitel går i dybden med brugen af SDN for at forbedre udnyttelsen af netværksressourcerne specifikt i datacentre. Miljøet i datacentre indeholder hybride pakke- og kredsløbsbaserede arkitekturer, på hvilke der kan anvendes intelligente algoritmer med henblik på at udvælge datatrafik, som med fordel kan offloades fra det kapacitetsudfordrede pakkebaserede netværk til optiske kredsløb.

På trods af at der er plads til forbedringer i forskningen, der danner baggrund for denne afhandling, så er der gennem kapitler identificeret og gennemgået kritiske aspekter ved SDN baseret QoS provisionering. Herudover er udført en lang række tests og demonstrationer ved hjælp af virtualiseringsteknikker. Disse test har til

formål understøtte de foreslåede ideer samt at danne et mere præcist billede af de praktiske anvendelsesmuligheder for SDN inkluderende besværlighederne ved brug i virtualiserede miljøer.

Acknowledgements

This thesis represents not only some years of research and hard work, but also the biggest project and challenge of my life so far. My experience at DTU and the PhD road overall have been nothing short of fulfilling and I have some remarkable individuals to thank for this.

First and foremost I would like to acknowledge the support of my thesis supervisors, José Soler and Lars Dittmann. Their guidance was invaluable for me during my PhD studies and I greatly appreciate them sharing knowledge and energy to help me have a great academic performance and best results.

To José Soler, I would like to state my gratitude, for his close supervising and involvement – he was there when I needed the incentives (however harsh) to keep going. Our working together has helped enormously to my self-development and to gaining valuable business acumen and has turned towards the end into a valuable relationship which helped me grow in many ways:

"Men are born for each other: so they either teach or tolerate." -MARCUS AURELIUS

I would like to also thank to all DTU and external partners involved in the COSIGN project. I am grateful to IBM Israel, and especially to Yaniv Ben-Itzhak, for their warm welcome, openness and amazing collaboration.

To my colleagues at DTU – Aleksander Siandy, Jahanzeb Farooq, Justas Poderys and Angelos Mimidis-Kentis, thanks for their companionship, supportive words and great days at DTU. Special thanks to Henrik Lehrmann Christiansen for proofreading this thesis and for his in depth feedback, and also to Henrik Wessing for helping with translating the abstract to danish.

I would also like to send acknowledgments back home, where the inherent source of life energy resides – to my family. Nothing could have been completed without the unconditional spiritual and material support and love, shadowing me since forever. Special thanks to my grandmother, whose thoughts always look after me and for all her wisdom, which I hope one day to achieve.

Last but not least, I would like to thank my (soon to be) fiancée, Roxana, for her supportive words, patience and love which have accompanied me during the good and bad days throughout the PhD.

Contents

x

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interfaces
A-CPI	Application-to-Controller Plane Interface
BBU	BaseBand Unit
BoD	Bandwidth on Demand
BS	Base Station
C-RAN	Cloud-Radio Access Network
CAPEX	Capital Expenditures
CBR	Constrained Based Routing/ Constant Bit Rate
CDT	Clamp Down Timer
CLI	Command Line Interface
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CRUD	Create Read Update Delete
Db	Database
D-CPI	Data-to-Controller Plane Interface
DC	Data Center
DCN	Data Center Network
DCNO	Data Center Network Optimization
DNF	Disjunctive Normal Form
DOT	Distributed Openflow Testbed
DPID	DataPath Identifier
DSCP	Differentiated Services Code Point
eNodeB	Evolved Node B
EDb	Electrical Db
EFD	Elephant Flow Detector
EPC	Evolved Packet Core
EPS	Electrical Packet Switched
EUTRAN	Evolved Universal Terrestrial RAN
FBFly	Flattened ButterFly
GPRS	General Packet Radio Service

GTP	GPRS Tunneling Protocol
GW	Gateway
ID	Identifier
IM	Information Model
IP	Internet Protocol
IT	Information Technology
KPI	Key Performance Indicator
LAR	Load Aware Routing
LTE	Long Term Evolution
MAC	Medium Access Control
MME	Mobility Management Entity
MNO	Mobile Network Operator
MPLS	Multi Protocol Label Switching
MVNO	Mobile Virtual Network Operator
NaaS	Network-as-a-Service
NB API	Nothbound API
NEMO	Network Modelling Language
NETCONF	Network Configuration Protocol
NNP	Non-Network Performance
NO	Network Observer
NP	Network Performance
NS	Network Scheduler
NSP	Network Service Provider
OCS	Optical Circuit Switched
ODb	Optical Db
ODL	OpenDaylight
OF	Openflow
OPEX	Operational Expenditures
OVS	Open vSwitch
OVS Db	Open vSwitch Database
OVSDB	Open vSwitch Database Management Protocol
OTT	Over-the-Top
PBNM	Policy Based Network Management
PC	Path Computation
PDN-DW	Packet Data Network Gateway
PM	Policy Manager
QoS	Quality of Services
RAN	Radio Access Network
REST	Representational State Transfer
RIB	Routing Information Base
ROA	Resource Orchestration Algorithm
RP	Resource Provisioning

RRH	Remote Radio Head
RTT	Round Trip Time
S-GW	Serving Gateway
SDN	Software Defined Networking
SDNC	Software Defined Networking Controller
SLA	Service Level Agreement
SPF	Shortest Path First
SQL	Structured Query Language
SRAM	Static Random-Access Memory
SSH	Secure Shell
SWP	Shortest Widest Path
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TE	Traffic Engineering
ToR	Top of the Rack
UE	User Equipment
vCPU	virtual CPU
VC	Virtual Circuit
VCID	Virtual Circuit Identifier
VLAN	Virtual Local Area Network
VM	Virtual Machine
VPN	Virtual Private Network
WAN	Wide Area Network

Publications in the thesis

Paper A

T. Galinac Grbac, **C. Caba**, J. Soler.
"Software Defined Networking demands on software technologies".
Published in: *Proceedings of 38th International Convention on Information and Communication Technology*, Opatija, Croatia, 2015.

Paper B

C. Caba, J. Soler.
"SDN-Based QoS Aware Network Service Provisioning".
Published in: *Springer Lecture Notes in Computer Science*, Vol. 9395, pp. 119-133, 2015.
Presented at: *International Conference on Mobile, Secure, and Programmable Networking*, Paris, France, 2015.

Paper C

C. Caba, J. Soler.
"APIs for QoS configuration in Software Defined Networks".
Published in: *Proceedings of 1st IEEE Conference on Network Softwarization (Netsoft)*, London, UK, 2015.

Paper D

A. Mimidis, **C. Caba**, J. Soler.
"Dynamic aggregation of traffic flows in SDN".
Published in: *Proceedings of 2nd IEEE Conference on Network Softwarization (Netsoft*, Seoul, Korea, 2016.

Paper E

C. Caba, J. Soler.
"Mitigating SDN controller performance bottlenecks".
Published in: *Proceedings of 24th International Conference on Computer Communication and Networks*, Las Vegas, Nevada, US, 2015.

Paper F

C. Caba, J. Soler.
"Mitigating the controller performance bottlenecks in Software Defined Networks".
Accepted for publication in: *International Journal of Communication Networks and Distributed System, Special Issue on: Software Defined Networks and Infrastructures, Network Function Virtualisation, Autonomous Systems and Network Management*.

Paper G

M. Artuso, **C. Caba**, H. L. Christiansen, J. Soler.
"Towards Flexbile SDN-based Management for Cloud-Based Mobile Networks".
Published in: *Proceedings of IEEE/IFIP Network Operations and Management Symposium* , Istanbul, Turkey, 2016.

Paper H

C. Caba, A, Mimidis, J. Soler.
"Model-Driven Policy Framework for Data Centers".
Accepted for presentation and to be published in: *Proceedings of 5th IEEE International Conference on Cloud Networking*, Pisa, Italy, 2016 .

Abstract I

Y. Ben-Itzhak, **C. Caba**, J. Soler.
"Utilizing Optical Circuits in Hybrid Packet/Circuit Data-Center Networks".
Published in: *Proceedings of the 9th ACM International on Systems and Storage Conference*, Haifa, Israel, 2016.

Paper J

Y. Ben-Itzhak, **C. Caba**, Shay Vargaftik
"C-Share : Optical Circuits Sharing for Software-Defined Data-Centers".
Submitted to: *HotNets 2016: Fifteenth ACM Workshop on Hot Topics in Networks*, Atlanta, Georgia, USA, 2016.

CHAPTER 1

Introduction

1.1 Motivation

Communication networks are a fundamental component in todays society, enabling innovative services for end users (mobile or fixed customers). End users expect good quality for the services they purchase, at very competitive prices. To fulfill these needs, Network Service Providers (NSPs) must operate their communication networks so that the network services are delivered with predictable quality, and the available network resources are used as efficiently as possible. Figure 1.1 illustrates simple scenario in which an NSP operates a communication network and sells network services to customers.

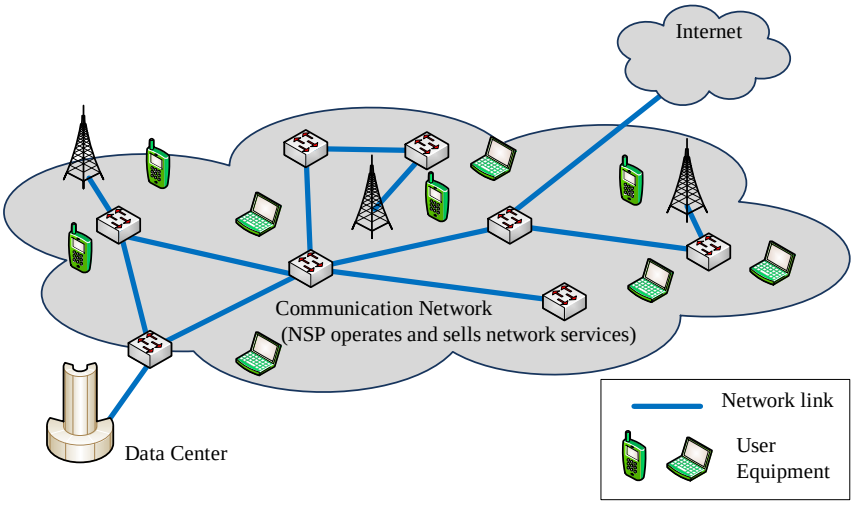


Figure 1.1: Network Service Providers operate a communication network and sell network services

Having predictable quality for a network service so that the needs of the user of the service are met, is termed *Quality of Services* (QoS) [1]. In communication networks, QoS depends on two types of factors. First, QoS depends on traditional *network performance parameters* such as bandwidth, delay, jitter, etc. Second, QoS

depends also on other Key Performance Indicators (KPIs) that are more customer centric and less oriented towards low level network performance mechanisms. A few examples from this category are the service provisioning time, service availability, and service repair time [1]. Hence, NSPs must carefully control all these aspects, belonging to both categories described here, in order to provide QoS in their communication infrastructure.

Deploying QoS mechanisms in communication networks requires complex configurations. Due to this reason, NSPs often rely on overprovisioning of network resources to cater to the needs of their customers. From an operational perspective it is easier to overprovision the network to assure that customers do not sense a degradation in their services, instead of configuring complex QoS mechanisms. Even though network overprovisioning satisfies the need for service quality at the current moment, it is unsustainable in the long term for the following reasons. By overprovisioning, the network will eventually reach the upper limits imposed by costs or by available technologies. This not only wastes network resources, but also forces the services to eventually compete for these resources in a best effort manner, leading to QoS degradation. Thus, good service quality is guaranteed only as long as the network has plenty of resources to avoid congestion. Ultimately, when congestion arises in the network, service quality will degrade since no QoS mechanisms are employed to assure network service differentiation.[2] For these reasons, overprovisioning the network is not sustainable. In order to counteract this trend of overprovisioning the network instead of applying proper QoS techniques, the development of network technologies must focus on simplifying the QoS provisioning mechanisms, and make them more accessible to NSPs.

1.2 Quality of Services in communication networks

As suggested previously, QoS in communication networks depends on several factors. Some of these factors are defined from the NSP's perspective and refer to low level network mechanisms. These are called *Network Performance* (NP) parameters and as example there are bandwidth, delay and jitter. Other factors are defined with a more customer centric perspective and they are referred to as *Non-Network Performance* (NNP) parameters in here. An example from this category is the service provisioning time, which depends on how fast the service request is processed, speed of network resources provisioning, etc. Communication networks should be designed to incorporate control mechanisms for all these factors in order to provide QoS.

Communication networks comprise functionality that can be divided in three logical planes: *data*, *control*, and *management* plane. Forwarding of user data occurs in the data plane. This implies various packet processing operations such as traffic shaping, policing, classification, queuing, etc. Forwarding of the data is controlled by mechanisms that are located in the control plane. Further, controlling the overall

network behavior, including control and data plane functions, belongs to the management plane. Examples of management plane operations are device configurations, service behavior specifications, and generally specifying policies for how the network should behave.

Controlling both the NP and NNP parameters towards achieving QoS in communication networks implies that several mechanisms are employed at all the three logical planes. In the data plane, packets must be processed according to their corresponding level of quality. Control plane mechanisms must operate towards achieving a good utilization of network resources, and few interruptions in service availability. Management plane operations must be fast in order to decrease the service provisioning time. Overall, the service provisioning should be automated to eliminate human intervention in order to increase the adoption QoS mechanisms in communication networks.

1.3 Software Defined Networking

Recent advances in network technologies aim to simplify the way networks are build and managed. Software Defined Networking (SDN) represents such a paradigm shift, which fundamentally changes the architecture of network devices and, as a result, of the entire communication network. Specifically, SDN promotes the decoupling of the control and data planes in network devices. This decoupling allows for the control logic to be centralized in an entity named *SDN Controller* (SDNC). By doing this, the network devices become simple forwarding elements which process the data packets as instructed by the logic residing in the SDNC.

Traditional network architectures have distributed control plane logic that is well integrated and tightly coupled with the data plane functions in the network devices. Management operations for traditional network devices are usually performed through Command Line Interface (CLI) configuration. In this respect, SDN brings significant changes by allowing the the SDNC to programmatically control the behavior of the forwarding devices. Moreover, by centralizing the control logic, the SDNC has a global view of the entire network allowing it to utilize the data plane resources more efficiently than by having only local knowledge as traditional network devices do. By leveraging the programmability and the global network view, SDN can simplify network operations and also increase the network resource utilization.

1.4 Problem statement and research directions

To encourage NSPs to deploy complex QoS mechanisms in their network instead of overprovisioning them, new network technologies must be developed to simplify QoS provisioning. If the QoS provisioning becomes simpler, then it can scale and be applicable in large and complex network environments. From a holistic perspective,

QoS is impacted by a wide range of parameters (discussed in section 1.2), which reflect in all three logical network planes: data, control and management. Hence, the pursuit of new network technologies to simplify QoS provisioning must consider several aspects that belong to all network planes.

SDN brings two benefits which are particularly relevant for the work in this thesis. First, SDN has the potential to simplify network management through programmability, since various operations can be easily automated. Second, the centralized control logic in SDN allows for a more efficient resource utilization, in comparison with traditional network architecture, due to its global overview of the network resources.

In this context, the research question for this thesis is formulated as follows:

Can SDN be applied to make QoS provisioning more scalable, so that it can be deployed in large and complex networks? In particular, is QoS provisioning simplified and further improved by using SDN techniques?

To answer the questions stated above, the research in this thesis aims to leverage the benefits brought by the novel SDN paradigm to improve QoS provisioning for network services. In this respect, the thesis initially investigates the challenge of QoS aware service provisioning in SDN. Further, the thesis takes a holistic approach with respect to QoS, by exploring what are the main challenges that SDN encounters at each of the logical network planes (i.e. data, control and management), with respect to scaling QoS provisioning to large networks. This holistic approach takes into account the entire spectrum of parameters that can impact QoS provisioning as described in section 1.2. In the final part of the thesis, the focus of the research is on leveraging SDN to obtain better resource utilization in communication networks and improve QoS for network services.

1.5 Thesis organization

The thesis is organized by closely following the research problem outlined above, so that each of the following chapters investigates a certain aspect of the problem. An initial introduction to SDN will be given in chapter 2, and all the content in this thesis will refer to the basic architectural definitions presented there, such as SDN architectural planes, interfaces, etc. Continuing, chapter 3 will investigate the challenge of QoS aware service provisioning in OF based SDN, by proposing a complete SDN architecture for QoS provisioning. After outlining the service provisioning architecture in chapter 3, the following chapters will investigate specific challenges at every layer of the SDN architecture, that can impede QoS provisioning. Chapter 4 will explore one of the main challenges raised at the SDN data plane with respect to QoS. This challenge is to overcome resources exhaustion in the data plane, which is an obstacle for QoS provisioning in large networks. Chapter 5 will focus on the control plane in SDN. Great effort is being put on increasing the performance of the control plane, which can

easily degrade when the underlying network grows very large. In this sense, chapter 5 will propose a novel approach to maintaining a good level of performance for the SDNC, in cases where the network scales up. The research that will be described chapter 6 focuses on network management plane challenges for QoS provisioning in SDN. In particular, this chapter will present a solution that aims to simplify network management operations, specifically for QoS provisioning, by enabling high level network policies. Finally, chapter 7 will describe the research work in the area of traffic engineering and optimization, applied to an SDN environment which is the most demanding nowadays: data centers.

It is important to mention that the references in this thesis are organized per chapter, so that each chapter has a corresponding section in the combined bibliography at the end of the thesis. The references indexing is restarted at each chapter.

CHAPTER 2

Introduction to Software Defined Networking

2.1 Short introduction to SDN

As mentioned in the previous chapter, SDN introduces a decoupled architecture in which the control plane logic is centralized in an entity named SDN Controller (SDNC), while the data plane remains distributed as simple forwarding elements. This split architecture, illustrated in Figure 2.1, comprises three logical planes (also termed layers): data plane, controller plane and application plane. A short overview of the architecture is given in the following.

A key component of the SDN architecture is the interface between the data plane, comprising the forwarding devices, and the SDNC (Figure 2.1). This interface is termed *Data to Controller Plane Interface* (D-CPI) [4]. Most commonly used protocol on the D-CPI to realize the SDN architecture is OpenFlow (OF) [3]. Through OF, the SDNC instructs the forwarding devices on how to process the data packets, as long as the devices support the OF protocol. The work done in this thesis is almost entirely based on OF version 1.0 (v1.0), hence, the following chapters are based on OF v1.0, unless stated otherwise.

Inside the SDNC, the control logic is often divided into several modules, each of them having a well defined purpose (Figure 2.1). Moreover, these modules offer services to every other module in the SDNC, building on top of each other to create a complete control plane logic. Figure 2.1 shows a generic controller architecture, however representative for many of the existing SDNC implementations [2] [6]. At the lowest level in the SDNC, there is the *OF* module, which implements the OF protocol to realize the communication with the data plane at the D-CPI. Another typical module in existing SDNCs is the *Topology Manager*, which maintains the logical representation of the underlying data plane infrastructure. This logical topology representation is created by leveraging the OF protocol to discover the links between the OF forwarding devices. Other modules are usually built over the basic features exposed by these two modules described above. For example, the *Forwarding* module in the figure instructs the data plane devices on how to forward traffic flows, by sending OF commands to the devices thus by using the *OF* module. Additionally, the *Routing* module uses the logical topology representation, maintained by the Topology Manager, to compute

routes for the traffic flows in the network. Further, the Routing module uses the features exposed by the Forwarding module to configure the computed routes in the data plane. Other examples that are depicted in Figure 2.1 are a *Firewall* module and a module that can provide Bandwidth on Demand (BoD). Given that some of these modules provide more advanced functionality such as firewalling and bandwidth reservation, they are termed *SDN services*. Essentially they provide certain network services in an SDN architecture. More SDN services can be created inside a controller, based on the features exposed by existing SDNC modules.

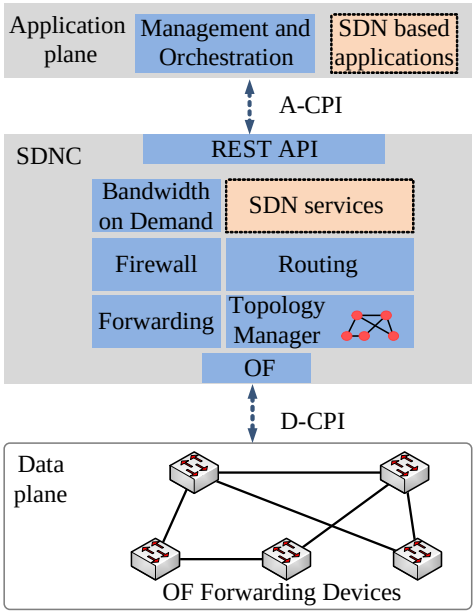


Figure 2.1: SDN architecture

The application plane comprises *SDN based applications* that utilize the functionality provided by the SDNC. This interaction between the SDN based applications and the controller is enabled through the *Application to Controller Plane Interface* (A-CPI). A-CPI is commonly realized in existing SDN implementations as Representational State Transfer (REST) Application Programming Interfaces (APIs) (Figure 2.1) [1]. Figure 2.1 depicts an example of an SDN application for management and orchestration of network services. This application would be targeted to administrators, but can also expose its own APIs for other entities to interact with.

As a general remark, an important distinction between traditional network architectures and SDN is in the organization of the data, control and management plane functionality. The data plane is clearly defined in SDN and contains packet forwarding

functionality, similar to traditional network devices (Figure 2.1). However, the control and management plane functionality is not clearly separated and is commonly implemented as specific modules in the SDNC, or as SDN applications in the application plane. In such cases, the SDNC becomes a platform that contains the control plain logic, but also additional functionality, even network management functions that are built on top of the control plane logic. For this reason, the terminology used to denote the SDNC is *controller plane* and not *control plane*.

2.2 Short introduction to OpenFlow

OpenFlow (OF) [3] is the main protocol used in the SDN architecture to realize the decoupling between the data and the control planes, and it is also the basis for the research work in this thesis. In consequence, this section provides a short introduction to cover the principles of OF.

Figure 2.2 shows the interaction between one OF forwarding device and the SDNC. OF communication at the D-CPI is realized by the *OF Agent* in the forwarding device and the OF module inside the SDNC. Apart from this, the OF protocol specification comprises also the flow table in the forwarding device, which is similar to a forwarding table in a traditional network switch. Specifically, the flow table contains entries that dictate how traffic is processed within the device. These entries in the flow table are termed *OF entries* or *OF rules*, since they represent packet processing rules. An OF entry contains a set of *packet header values*, a set of *actions* and various *counters*. When a new packet arrives at the forwarding device, the headers of the packet are matched against the header values defined in each OF entry in the flow table. Further, the OF rule with the longest match is selected, and the actions comprised by this OF rule are applied to the packet.

In OF v1.0, the packet headers that can be considered for matching are according to the Ethernet based TCP/IP communication. As a result, a packet can be matched based on Ethernet address field, the EtherType field of the Ethernet header, IP v4 addresses, Transport layer ports, etc. [3]. Depending on the chosen header fields, the granularity of the matching varies from coarse granular, matching on Ethernet headers, to more fine granular, by matching on Transport layer headers. Some of the most common actions in version 1.0 of OF are: *drop* - which indicates that the packet must be dropped, *output* - which indicates that a packet should be sent out a certain port, and *enqueue* - which indicates both the output port and the queue number. Using these matching headers and actions, the SDNC can control the traffic forwarding in the data plane by installing OF entries in the forwarding devices. The OF message used to install OF entries in a forwarding device is termed *FlowMod*.

With respect to forwarding, the SDNC can exhibit two types of behavior. The *proactive* OF behavior implies that the SDNC pre-installs OF entries in the forwarding devices so that when a new packet arrives, there is already a corresponding OF rule

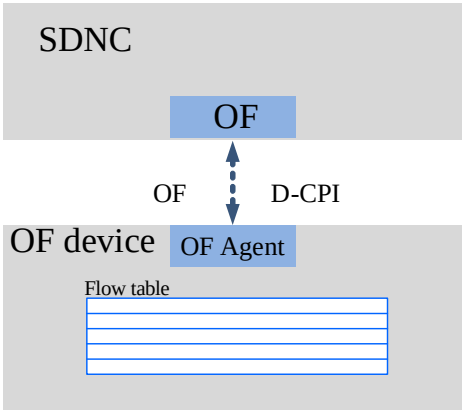


Figure 2.2: OpenFlow based SDN setup

to control how the packet should be processed. However, there is also the *reactive* OF behavior. This reactive behavior occurs when an arriving packet has no matching OF rule in a forwarding device. In this case, the packet is encapsulated in an OF *Packet In* message and sent to the SDNC. At the SDNC, the packet is processed and a new OF rule is installed in the corresponding forwarding device to match the specific traffic flow indicated by the arriving packet.

Another example of OF message is *Statistics Request*. By using this type of message, the SDNC can poll the counters in the forwarding device to get various statistics. For example, the SDNC can poll the OF entries counters and compute how many packets have matched a certain OF entry. Moreover, the SDN can use this type of message to get flow table level statistics (e.g. number of active OF entries), or port level statistics (e.g. sent bytes).

Up to date, there are many types of forwarding devices that support OF v1.0, and also more recent versions of the protocol. There has been significant effort put into developing tools for experimenting with OF based SDN. In this sense, it is possible to emulate real OF devices by using software switches. A well known OF software switch in the SDN community is Open vSwitch (OVS) [5]. The research performed in this thesis uses OVS as an OF forwarding device for the various testbeds designed in each chapter.

Based on this initial introduction into OF based SDN, the following sections will approach the challenge of providing QoS aware network service in SDN. Other details of the OF protocol and SDN behavior are gradually introduced through the thesis, to account for the specific context and problem where OF is used.

CHAPTER 3

QoS-aware service provisioning in SDN

Software Defined Networking (SDN) promotes separating the data and control planes in network devices, which represents a significant shift from traditional networks. However, the new SDN architecture can be used to provide similar services to what existing traditional network architectures provide. Due to new protocols, especially Openflow (OF), services are provisioned through different mechanisms in SDN. This chapter describes how network services are provisioned in an OF based SDN. Specifically, section 3.1 focuses on enabling QoS for the SDN based services. To illustrate the new model for QoS aware service provisioning, a simple network service is taken as an example. After the mechanism for service provisioning is formally described, an emulated OF environment is configured and used to validate the proposed solution. While OF allows for controlling the traffic in the network, it is not sufficient for providing complex QoS services, which require more advanced configuration of network resources. The challenge of simplifying QoS configuration in SDN is addressed in section 3.2. Overall, the current chapter draws a complete picture about SDN based service provisioning with predictable QoS. Once a clear picture is illustrated, the following chapters of the thesis proceed with exploring the challenges that arise at each plane of the SDN architecture, that could potentially obstruct QoS.

The work presented in this chapter is based on two papers. Section 3.1 is based on Paper B and section 3.2 is based on Paper C.

3.1 Provisioning of QoS-aware network services

To grasp the benefits of SDN and OF for QoS, it is important to first understand the mechanisms by which basic network services are provisioned. As also hinted in chapter 2, creating network services in SDN implies controlling the logical resources maintained by the SDN Controller (SDNC), which are the logical counterpart of the physical network resources. This section focuses on the management of network resources, in the context of SDN based service provisioning with predictable QoS. A simple network service, *Virtual Circuit* (VC), is taken as an example to illustrate the mechanisms.

Various techniques can be used to deploy QoS in IP networks. One such technique

is Diffserv-aware MPLS-TE, which is based on combining the Diffserv data plane architecture with the MPLS-TE control plane [12]. An important benefit of using DiffServ is scalability, due to its per hop behavior, for which nodes perform traffic conditioning independently of other nodes, thus in a stateless manner. Additionally, the DiffServ-aware MPLS-TE solution manages to avoid congestion in the network due to its constraint based path computation algorithm, which is characteristic to MPLS-TE control plane.

A similar approach to the one described above is used in this section to provide network services with predictable QoS in the SDN architecture. Specifically, the QoS mechanism applied in the data plane is based on DiffServ: applying traffic conditioning and admission control at the ingress and per hop behavior according to traffic classes in the core of the network [33]. For the control plane, a Constrained Based Routing (CBR) algorithm is proposed to compute paths in the network based on various QoS constraints such as bandwidth, delay, etc. Instead of using the distributed MPLS control plane as proposed in [12], the centralized SDN architecture is used here.

In this work, the following scenario is considered: a network service provider offers VCs as services to its customers, which can be smaller service providers or other enterprise customers that need network connectivity (Figure 3.1). In this context, a *VC* is defined as a stream of packets, comprising one or more traffic flows, that are identified as a single unit (aggregated) and are processed uniformly in the network. Based on the negotiated Service Level Agreement (SLA), the VC has a set of associated parameters such as *start time*, *duration*, a *QoS profile*, and other path constraints. Thus, the SDN-based architecture in the provider's network must ensure that the VCs are implemented so that they comply with the parameters specified in the SLA. Since the focus of the current thesis is QoS, the Network Performance (NP) parameters considered in this section are *bandwidth* and *delay*.

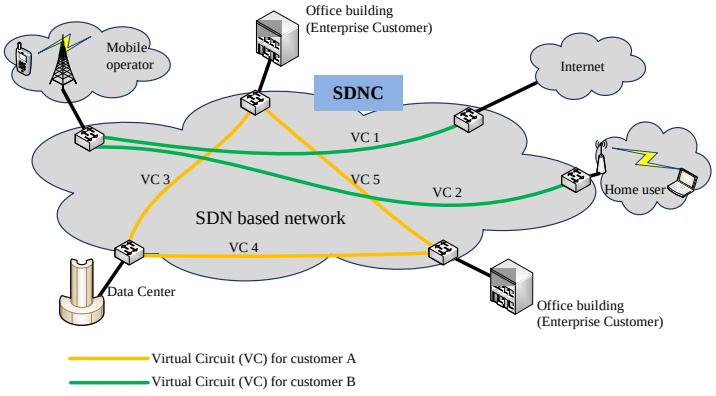


Figure 3.1: NSP offers Virtual Circuits as services in an SDN based network

3.1.1 Architecture

Figure 3.2 gives an overview of the control and data plane entities involved in the VC service provisioning. The architecture illustrated in this figure follows the same definitions as described in section 1.3. At the data plane there are the forwarding devices (e.g. D1, D2, etc.), which contain resources like ports and queues. QoS is enforced by configuring these resources in the data plane. The control plane in this architecture is denoted by the SDNC. At the core of the SDNC there is a representation of the *network resources*, here illustrated as an Information Model (IM). Resources that are allocated to a certain customer as part of a service offering are denoted as *customer resources*, while the totality of the underlying network resources is captured by the *network resources* portion of the IM. Apart from the IM, two types of functions are comprised by the SDNC:

- **Control functions** represent operations that must occur on short time scales, or even in almost real time (e.g. hours, minutes, seconds). Examples of such operations are orchestration of traffic flows, statistics gathering (monitoring), rapid QoS enforcement by dropping packets, etc. OF is the chosen protocol to implement all the control functions discussed in this thesis.

- **Management functions** implement their purpose by using a management protocol at the D-CPI. Since OF is a control protocol used to control traffic forwarding in the network, another protocol must be used to realize the management operations. These operations are often configuration of network resources, with a longer time scale (e.g. days) when compared to control functions.

In order to offer services with predictable QoS, the underlying network resources that form these service must be managed correctly to avoid congestion and provide quality differentiation. For this, two mechanisms must be in place (outlined in blue in the figure):

1. A QoS mechanism for the data plane comprising resources and possible configuration such that predictable QoS can be offered.

2. A control plane strategy for managing the resources, perform admission control, etc. This strategy is applied on the logical network resources kept by the SDNC and defined in the information model in Figure 3.2.

In the SDN architecture illustrated in Figure 3.2, each control and management function performs a set of operations to fulfill its purpose. These operations target the logical resources in the SDNC (represented by the IM). Such an operation may be translated into an operation on the physical network resources in the data plane through control or management protocols commands. As an example, an operation for adding more bandwidth to a circuit in the network first occurs in the SDNC by

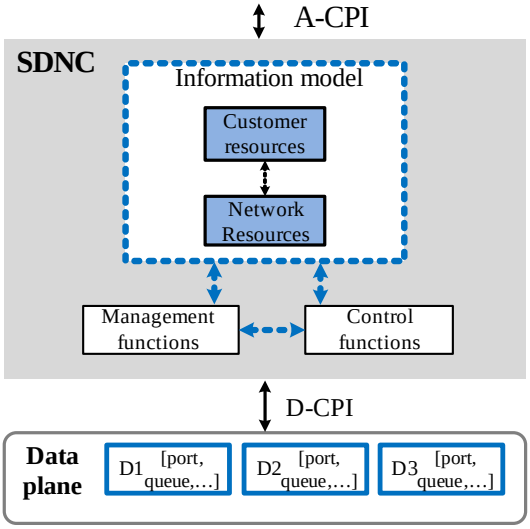


Figure 3.2: Generic architecture for QoS-aware service provisioning

updating the logical circuit and device resources accordingly. Then, the data plane forwarding devices (simply termed forwarding devices herein) are configured to reflect the modifications in their logical counter parts. In this manner, the control plane strategy for resource management enforces a particular behavior into the data plane resources in SDN. It can be inferred from here that both traffic control and resource configuration techniques are needed to correctly implement QoS mechanisms.

3.1.2 Related work

There is significant work done in the area of SDN based provisioning of network resources towards realizing services with predictable QoS. Some of the work targets automated, policy based network provisioning [5] [11], while other focuses on traffic engineering mechanisms across Wide Area Networks (WAN) [15] [8] [21] [18] [10]. Data centers are also a key area where dynamic allocation of network resources is needed, especially due to their strong requirements with respect to being flexible and agile. Hence, there are several studies that address this particular challenge. For example, [14] presents an algorithm for allocation of bandwidth resources between Virtual Machines (VMs) by using the OF protocol, while [6] describes a platform for network, compute and storage service provisioning in data centers.

Most of the related work addresses the high level logic for the QoS aware service provisioning, omitting first the mechanisms for resource management at the SDNC, and second, how the service logic is enforced in the data plane. To give an example,

in [18], the authors state that their solution relies on traffic classification and rate limiting at the network edge, without describing the details of how the management operations of the logical resources are enforced into the forwarding devices. These details are important for understanding the underlying principles of a QoS aware service provisioning platform, allowing to build on top of, and improve existing solutions.

The solution proposed in [19] resembles the one presented in this chapter. Their data plane QoS architecture for SDN is based on similar techniques such as combining rate limiting queues, however, having different constraints for these queues from the ones used herein, which will be described later. Unlike [19], the current work precisely defines how the logical resources are managed in the SDNC in order to provide deterministic QoS. Additionally, two models for resources reservation are described in this chapter, aiming to serve as a basis for understanding and building SDN frameworks for on demand bandwidth provisioning with specific QoS requirements.

Another paper describing closely related work is [1], which presents an SDN architecture for cloud services provisioning with QoS guarantees. Their solution also relies on traffic classification by employing priority queues at the output ports in the forwarding devices. Further, they have validated the proposal by using Open vSwitch (OVS), which is the same approach taken in this chapter. Despite these similarities, the work herein aims to provide a generic approach towards QoS aware network service provisioning, suitable for various other applications and not only for cloud services.

A novel concept that served as an inspiration is *Network-as-a-Service* (NaaS), which is discussed in various research papers [6] [29]. In NaaS, the network infrastructure is abstracted so that network services can be seamlessly provisioned end-to-end using simple request/response messages. This paradigm follows the already established provisioning models for IT resources (compute, storage). QoS is a common criterion when requesting a service in NaaS. [29] presents an SDN framework for NaaS which includes also various QoS criteria. Specifically, the authors describe an analytical model to provide end-to-end connectivity by taking into account NP parameters such as bandwidth and delay. Further, the authors show that in packet switched networks, the delay of a traffic flow can be bounded simply by allocating enough bandwidth for that flow. This chapter is based on the same argument. While this idea is intuitive, the challenge lies in allocating just enough bandwidth for a traffic flow in order to satisfy the QoS demands, and also increase resource utilization. This is contrary to the general overprovisioning approach which many service providers are taking.

This chapter presents a detailed solution for provisioning of network resources with QoS guarantees. The proposed solution considers both the data and control planes of the SDN architecture. Specifically, it precisely describes the data plane QoS mechanism, it presents the resources management at the control plane, and it also maps the control plane operations to corresponding data plane configurations. Moreover, the QoS mechanism is validated through a series of tests performed in a virtualized environment based on Mininet [20].

3.1.3 Data plane QoS control mechanism

In this chapter, a generic data plane QoS control mechanism will be described. This mechanism is not SDN specific, however it can be used in an SDN architecture to provide predictable QoS for the network services. Two types of traffic flows are considered here as being transported in the data plane: *best effort flows* and *QoS flows*. For best effort traffic flows there is no need to allocate network resources. Conversely, *QoS flows* require that resources are reserved such that the QoS for this type of flows is predictable. Being predictable means that the QoS is within certain bounds as negotiated in the SLA. Since best effort traffic does not require complex mechanisms at the control and data planes, the focus of this work is on the QoS traffic flows, even though best effort traffic is also considered.

Figure 3.3 shows the schema of a forwarding device and the logical entities that are involved in the QoS provisioning process. As it can be noticed, the proposed data plane QoS control mechanism uses rate limiters to perform admission control (traffic conditioning), and priority queues to prioritize among various traffic classes. Traffic flows admission control occurs at the ingress forwarding devices, according to the measured NP parameters and resource demands. SDNC monitors the network, decides whether a traffic flow can be admitted, and further enforces its decision by configuring the corresponding forwarding devices. Throughout the lifetime of the requested network service, the SDNC must maintain the data plane QoS configuration as presented further in this section through a set of equations. These equations are termed *invariants* herein, since they must always be valid (i.e. not vary), in order to have predictable QoS. By maintaining the invariants presented further, congestion in the output queues is avoided. As noted in [2], by controlling the congestion at the output port, the queuing delay experienced by the packets is bounded. By having bounded delay and flexible bandwidth allocation, a predictable level of QoS can be assured for network services.

The total capacity (i.e. rate) of each output port in a forwarding device is shared between several priority queues. There is one priority queue for best effort traffic flows (Q_b), and one queue for each QoS class (Q_1 ... Q_m) (Figure 3.3). Best effort traffic is served by Q_b, and the QoS traffic flows are served by the other queues according to the QoS class they are mapped to. Four QoS parameters are used here to characterize a priority queue: the minimum serving rate (R_{min}), the maximum serving rate (R_{max}), the queue size (S), and the queue priority. The scheduler serves each queue according to the configured rates (Figure 3.3).

Equation 3.1 describes the relation between the serving rates of a priority queue and the total port capacity (C_p):

$$R_{min} \leq R_Q \leq R_{max} \leq C_p \tag{3.1}$$

R_{min} represents the guaranteed rate for a queue. The actual rate of a queue (R_Q) is always at least the guaranteed rate, but it can increase up to R_{max}, if there is available

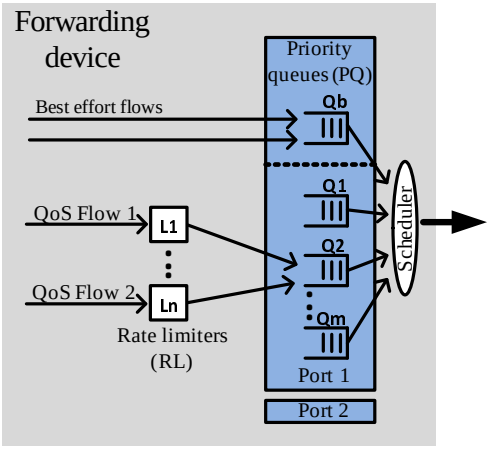


Figure 3.3: Conceptual model for the forwarding device and its logical entities

bandwidth at the output port (i.e. the port's capacity is shared).

All the best effort traffic flows share Q_b, hence, there is no per flow resource management (e.g. admission control) for the best effort class. In this case, the bandwidth associated with Q_b (transmission rate) may not be enough to serve all the best effort flows with a good level of QoS. This causes congestion, leading to a poor level of service for this traffic class (i.e. increased delay, packet drops, etc.).

There are two processing stages for the QoS flows (Figure 3.4). In the first stage, rate limiters are used to perform admission control for the incoming traffic flows, according to a set of token bucket parameters: average rate R_L, which is guaranteed for a flow (committed information rate), and bucket size B denoting the maximum burst size (excess information rate). The second stage contains the priority queues for the QoS classes (Q_1 ... Q_m). This set of queues is termed *QoS slice* herein, denoting the bandwidth slice for an output port that is allocated to QoS flows.

Consider a set of rate limiters, N, each applied to a QoS flow such that $R_{L,i}$ is the guaranteed rate for the QoS flow at rate limiter L_i. All the QoS flows associated with rate limiters in N, have the same QoS class, which is associated with queue Q_j. $R_{min,Qj}$ is the guaranteed serving rate of queue Q_j. M denotes the set of priority queues in the QoS slice (Q_1 ... Q_m). In this context, Equation 3.2 describes the resource allocation for Q_j, in relation to the rate limiters in set N.

$$\sum_{i \in N} R_{L,i} \leq R_{min,Qj}, \forall j \in M \tag{3.2}$$

Through the constraint specified in Equation 3.2 , the SDNC ensures that there is enough bandwidth in the priority queues belonging to the QoS slice in order to serve

the QoS flows with the minimum guaranteed rate (i.e. $R_{L,i}$). A QoS flow may also be served with a rate higher than the minimum guaranteed, as specified by the burst size (B) of the rate limiter. For this, the corresponding priority queue must have additional unused bandwidth. Through rate limiting and prioritization, the congestion at the output port can be controlled so that the QoS flows have predictable QoS in terms of bandwidth and delay bounds.

The bandwidth allocation for an entire output port is governed by the following equation, indicating that the sum of the guaranteed rates for all the queues at the output port ($Q_b, Q_1 \ldots Q_m$) cannot be higher than the capacity of the port. Apart from this, the excess available bandwidth (i.e. not reserved or used at a certain moment) at the output port is shared between all the queues (including the best effort queue). This sharing is done according to the priority and the $R_{min,Qj}$ configured for each queue. This behavior is also demonstrated further in the chapter through experiments.

$$\sum_{i \in M} R_{min,Qi} + R_{min,Qb} \leq C_p \tag{3.3}$$

SDNC manages the bandwidth sharing and the congestion for each output port, by configuring the resources according to the equations described above. This congestion control technique influences the upper bounds for the queuing delay. Besides, the end-to-end propagation delay for the traffic flows is reflected in the length of the path computed by the CBR inside the SDNC. Thus, SDNC has control over two major components in the end-to-end delay: queuing and propagation delay.

3.1.4 Prototype implementation

The prototype architecture is depicted in Figure 3.4. Mininet [20] has been used to emulate the data plane, which is based on OVS software switches [26]. It is possible to configure queues for traffic control at the output ports in OVS, with the following parameters: R_{min}, R_{max}, S and priority, all having the same semantics as described in 3.1.3. In the current work, the SDNC configures the QoS resources (i.e. priority queues) in the forwarding devices through the Open vSwitch Database Management (OVSDB) protocol [3]. Different protocols can be used, depending on the type of network devices, as long as they support configuration of QoS resources for those devices. For example, the Network Configuration protocol (NETCONF) [31] is an alternative management protocol. On the other hand, OF v1.0 [22] is used to control the traffic flow forwarding in the data plane. The entire framework is built as an extension to the open source SDNC platform Floodlight [13].

As it can be noticed, the proposed and implemented architecture is modular. A part of the modules illustrated in the figure are re-used from Floodlight (in blue), while the rest of them are newly proposed modules. The purpose of each module is as follows:

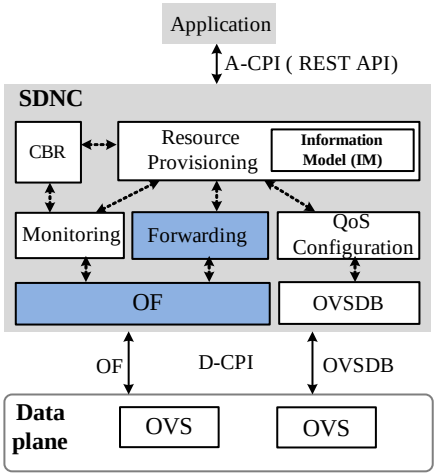


Figure 3.4: Architecture of the prototype showing the main functional entities

- *OF* and *OVSDB* modules: their purpose is to realize the communication with the data plane at the D-CPI through the OF and OVSDB protocols.
- *Monitoring* module: fetches statistics from the network and averages them in oder to alleviate the spikes in the measurements. This is to reduce the oscillations in the routing algorithm which is based on the measured network load.
- *CBR* module: it implements the Shortest Widest Path (SWP) algorithm as described in [34]. SWP consists of two steps: (1) it first choses a path to satisfy the bandwidth constraint, and (2) from the output of step 1 it choses the path with the lowest end-to-end delay.
- *Forwarding* module: implements the logic for pushing OF entries into the flow tables of the forwarding devices.
- *QoS Configuration* module: it comprises a set of APIs to configure QoS resources in the data plane, by leveraging the OVSDB protocol.
- *Resource Provisioning* (RP) module: comprises the logic for the VC provisioning service. RP module uses the functionality exposed by other SDNC modules to provide the required service. For instance, it uses the monitoring information from *Monitoring* module, it gets routes that satisfy QoS constraints from the *CBR* module , etc. Besides, RP comprises the IM showed in Figure 3.5,for which it must maintain a consistent view in order to avoid erroneous allocation of network resources. If the logical resources in the IM do not reflect the actual

physical network resources (i.e. inconsistencies exist), deficient allocation can occur which further leads to unwanted congestion in the network.

There are two types of resources in the IM in Figure 3.5. On the left side, there are the *customer resources*, which represent a subset of the total *network resources* (on the right side). In the *customer resources* model, each customer may have several VCs, which represent the network service they purchase. Further, each VC has a QoS descriptor to define its QoS characteristics, which can change over time if the customer demands a change in the VC service. The *network resources* model comprise the overall network topology, with nodes, ports, and queues.

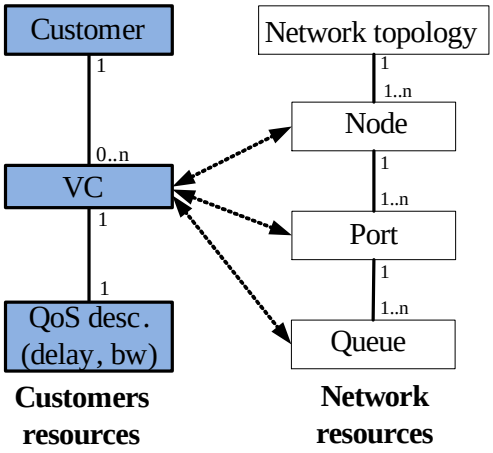


Figure 3.5: Information Model (IM)

The QoS characteristics of a VC are defined by two factors:

- A *path*, which contains a set of nodes and ports in the network. The quality of the path influences the QoS level of the VC for which the path has been computed.
- A traffic class (also named QoS class in 3.1.3), which denotes the *priority queue* used at the output port to forward the corresponding traffic for the VC.

3.1.5 The VC provisioning algorithm

The VC provisioning logic is explained by following the same scenario where a customer initiated a request for a VC with certain QoS characteristics (i.e. QoS profile) towards an NSP. This request is first processed by the management plane in the NSP's domain, which is out of the scope of this work. After, the request is processed by the SDNC,

which provisions the requested service. Two operational models for VC provisioning (i.e. Model A and B) are considered here, depending on which, the algorithm for provisioning VCs varies slightly. Both variations of the algorithm are presented next and illustrated in Figure 3.6.

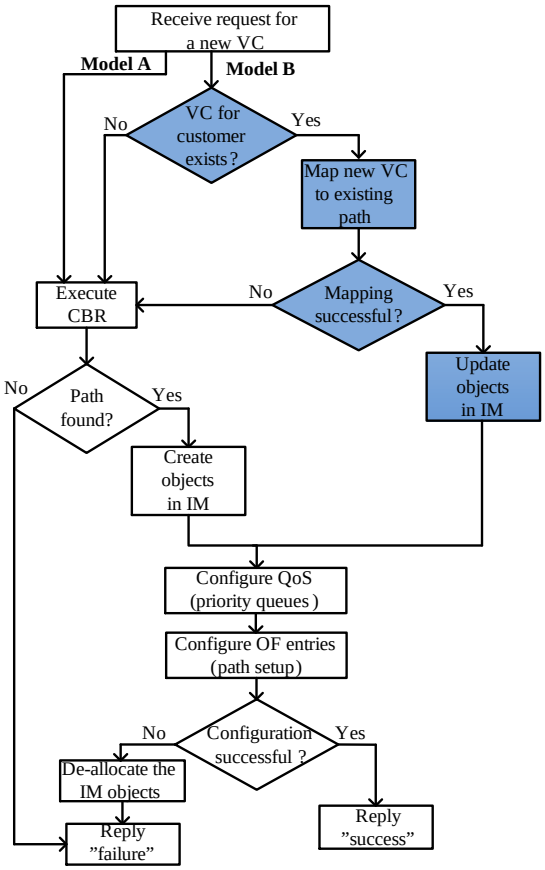


Figure 3.6: Combined VC provisioning algorithm for operational models A and B

Model A

Model A consists of fine granular VC provisioning, for which each VC has an individual QoS profile. This means that specific QoS network performance parameters are requested and must be offered for each VC. In this operational model, when a new VC request is received, the SDNC executes the CBR algorithm to compute a path that

satisfies the QoS constraints of the new VC. If no path is found to satisfy the QoS constraints, the request for service is rejected and a *Failure* reply is sent to the requestor. However, if the computation resulted with a feasible path for the new VC then the IM is updated to reflect the new resource allocation (Figure 3.6). Specifically, a VC object is created with a corresponding QoS descriptor, and associated with the requesting customer. Moreover, the network resources are updated as well by subtracting the newly allocated customer resources. Next, the RP module configures the QoS resources in the forwarding devices along the computed path. This configuration comprises rate limiters at the ingress devices and priority queues in the core and egress devices. To finalize the creation of the VC, OF rules are installed in the devices to enable end-to-end traffic forwarding. After these data plane configuration steps are executed successfully, the RP module replies with a *Success* response to the initial VC request. Conversely, if the data plane configuration fails due to various reasons, the IM state is reversed by de-allocating the resources, and a *Failure* reply is sent to the customer.

Model B

Model B assumes a coarser granular VC provisioning, where several VCs share a subset of the QoS profile parameters. Specifically, even though each request for a VC service has an individual QoS profile, more VCs may share the same path in the network, and only be differentiated among themselves by the traffic class they belong to. In Figure 3.6, the blue blocks denote the logic part which is particular only to this operational model. Upon receiving a new request for a VC, the RP module extracts the VC endpoints. Then, the RP module verifies if there is an already a VC established between these endpoints, and for the same customer. If there is no existing VC with these conditions, then the logic proceeds in the same manner as described for *Model A*. Alternatively, if a VC already exists, the RP module tries to map the newly requested VC on the path of the existing VC. This process takes into account the QoS characteristics and the availability of the data plane resources (i.e. bandwidth at the output port) along the existing path. Thus, if the mapping is successful, the new VC reuses the same path as the existing VC. However, different priority queues may be used at each output port, reflecting a difference in the traffic class (i.e. QoS characteristics) between the existing VC and the new VC. If the mapping is possible, the RP module updates the IM accordingly: the existing VC is enhanced with more resources and potentially an additional set of queues, denoting two different traffic classes multiplexed into the same VC. After the updates are made, the logic continues as described for *Model A*.

Comparison between the two models

Both operational models have benefits and drawbacks making them suitable for different scenarios. The main difference is due to the way VCs are managed for a customer. In *Model A*, each VC for a customer has its own path and and traffic class.

This gives more accurate control over the QoS characteristics of the VCs, ensuring a better QoS isolation (no resource sharing among VCs). In contrast, for *Model B*, several VCs belonging to a customer may share the same path. In consequence, even if individual VCs may be mapped to different traffic classes (i.e. priority queues), they are still treated as a single unit when the traffic is forwarded along the network path. This decreases the flexibility in some cases such as when the VCs must be rerouted to a new path, for traffic engineering purposes. In case a rerouting is needed, the new path must satisfy more complex QoS constraints, according to the multiple interdependent VCs that share the same path.

As a result, *operational model A* is more suitable for scenarios where good QoS isolation between VCs is mandatory, or where stringent SLAs must be satisfied. An example of such scenario can be provisioning of VCs for transport over packet carrier networks using IP/MPLS. On the other hand, *operational model B* is more suitable for scenarios where a customer purchases connectivity between two endpoints for different traffic types (QoS classes), possibly required by various applications that the customer uses. In this latter scenario, strict QoS isolation between the application traffic flows multiplexed in the same path (i.e. VC) may not be mandatory. In addition, the advantage of *Model B* over *Model A* is that the computationally intensive CBR algorithm can be avoided for some of the VC requests, hence reducing the initial delay for the flow setup (Figure 3.6).

3.1.6 Validation of QoS mechanism

In order to apply the proposed solution for QoS aware service provisioning in SDN, it is necessary that the SDN architecture supports the proposed mechanisms at the level of both the data and the control plane. In section 3.1.3, the data plane QoS control mechanism has been formally described, while in sections 3.1.4 and 3.1.5 the focus was on the control plane components and logic. This section proceeds to validate that the proposed data plane QoS mechanism can be implemented in OF devices. By doing this, the chapter aims to prove that the proposed QoS mechanism can be the basis for QoS aware service provisioning in an SDN architecture. For this purpose, the Mininet testbed illustrated in Figure 3.7 is used. More precisely, the purpose of the tests performed in this section is to verify and prove that the OVS instances, which are the basis of Mininet, can implement the QoS mechanism described in section 3.1.3.

A simple network topology is enough to illustrate the behavior of the OVS software switches, which represent the forwarding devices with respect to the generic SDN architecture described earlier in the chapter. Hence, the testbed contains only two interconnected OVS switches, each of them having a host attached to it, which serve as a traffic source or destination (Figure 3.7). Both switches (S1 and S2) have three queues at their output ports (i.e. Q_1, Q_2 and Q_3). Queue configuration is as represented in the figure, with the parameters R_{min}, R_{max}, and *priority*. The *size* parameter for the queues is not relevant since it impacts only the queuing delay, which

out of the scope of the validation. It is also important to mention that for the tests performed here, the network link have been dimensioned to 5 Mbps, meaning that the capacity of the output ports in the OVS instances is 5 Mbps.

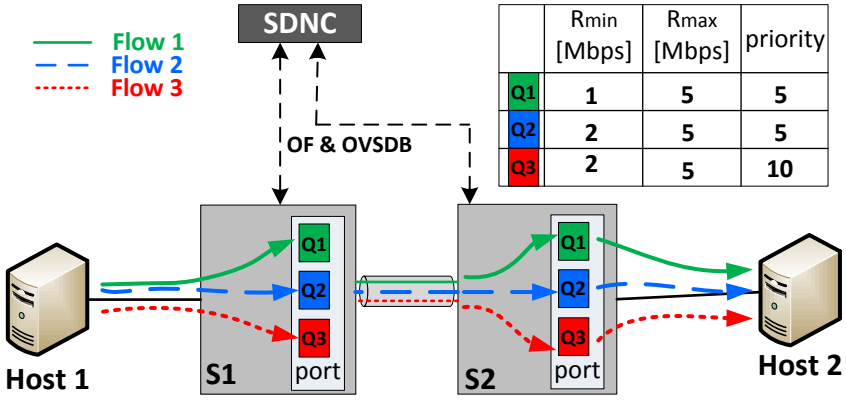


Figure 3.7: OF testbed used to validate the QoS mechanism

During the tests, the management of the queues at the output port (i.e. creation, modification, deletion) has been done by using the features exposed by the *QoS Configuration* module. This module translates high level commands into OVSDB protocol messages. In SDN, QoS configuration is an important aspect since it complements the OF protocol in providing QoS aware networks services. Consequently, it is equally important to study the QoS configuration as it is to investigate OF based solutions to implement SDN services. While this section proposes and demonstrates a framework for QoS aware SDN based service provisioning, the next section studies the QoS configuration aspect in depth.

For the tests, three TCP traffic flows (flow1, flow2, and flow3) are transmitted from *Host 1* to *Host 2*. The traffic flows have been generated using Iperf [16]. Each of the traffic flows has a rate of 10 Mbps when transmitted, and it is mapped to one of the queues so that *flow 1* corresponds to Q_1, *flow 2* corresponds to Q_2, and *flow 3* corresponds to Q_3. Even though these flows do not represent realistic network traffic, they are useful to drive the rate limiting queues in order to emphasize their behavior. By the same token, the transmission rate of the flows (i.e. 10 Mbps) is much higher than the link capacity and the R_{max} (i.e. 5 Mbps) in order to saturate the queues. In this setup, by investigating the patterns in the throughput for each of the flows, it is possible to characterize the behavior of the priority queues and

verify if it corresponds to their configuration and to the expected behavior, presented in section 3.1.3. Specifically, the most interesting characteristic to investigate is how the available bandwidth is shared among the queues, since this can assure QoS differentiation among different traffic classes.

Test 1

For the first test, each traffic flow is transmitted one at a time, thus utilizing the entire link capacity without competing for bandwidth with any other flow. After, the throughput for the flows has been measured and averaged at the receiving end. The obtained throughput measurements are depicted in Figure 3.8, together with the standard deviation. As it can be noticed, the average throughput recorded for each traffic flow is approximately 4 Mbps. Even though the queues are configured with guaranteed rates of 1 or 2 Mbps (Figure 3.8), they serve the flows with a rate of almost the entire link capacity. This behavior is due to the maximum serving rate that is configured for the queues, which is 5 Mbps. However, the actual measured throughput is approximately 80% of the maximum possible capacity. Some of the reasons for this reduced measured throughput could be that the reports of the traffic generation tool do not include the lower layer protocol overhead (e.g. Ethernet) and the retransmissions in the network. Also, if the default window size for the TCP flow control mechanism is not properly adjusted, the utilization of the links can be sub optimal [30]. The obtained results confirm that a priority queue in OVS serves its traffic with a rate up to R_{max}, if there is available bandwidth at the output port, hence, utilizing the entire available capacity at the output port.

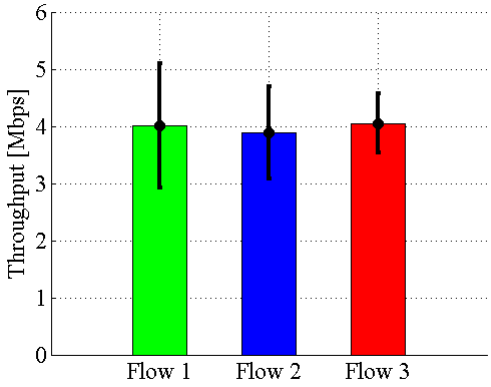
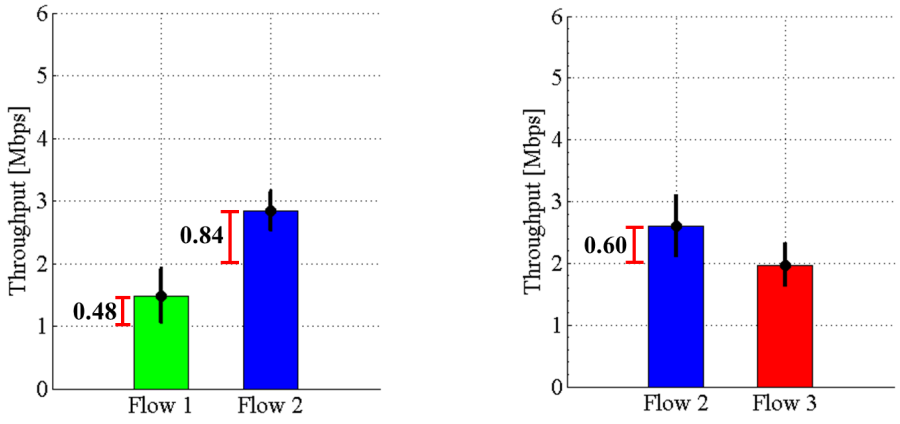


Figure 3.8: Average throughput results for test 1

Test 2

For the second test, *flow 1* and *flow 2* are transmitted at the same time, while *flow 3* is not transmitted. This is to specifically investigate the bandwidth sharing between same priority queues, because Q_1 and Q_2 have both a priority of 5 (Figure 3.7). According to the configured R_{min}, Q_1 should have 1 Mbps guaranteed rate and Q_2 should have 2 Mbps, summing up to 3 Mbps. Since the port capacity is 5 Mbps, there is an excess of 2 Mbps (theoretically) when subtracting the guaranteed rates for Q_1 and Q_2. These 2 Mbps are shared between the two queues so that Q_1 receives 0.48 Mbps and Q_2 receives 0.84 Mbps, in addition to the guaranteed bandwidth. This can be noticed in the results in Figure 3.9a. Even though both queues have the same priority, the extra bandwidth is not shared in equal proportions, according to the priority. Specifically, Q_2 receives almost twice more bandwidth than Q_1. This suggests that in OVS, the configured guaranteed rates are also considered in the bandwidth sharing algorithm at the output port. As a result, each queue receives extra bandwidth proportionally to their R_{min} parameter, considering that they have equal priorities (also depicted in the figure).



(a) Test 2

(b) Test 3

Figure 3.9: Average throughput results

Test 3

For the third test, only *flow 2* and *flow 3* are transmitted simultaneously, in order to assess the bandwidth sharing between queues with different priorities. Specifically, Q_2 with priority 5 and Q_3 with priority 10. The results for this test are showed in

Figure 3.9b. In this case, both queues receive the guaranteed rate of 2 Mbps. while the excess bandwidth is entirely allocated to the queue with the highest priority (i.e. Q_2). With the risk of being confusing, the queue configuration in this work closely follows the OVS specifications, so that the queue with a lower priority value is more important. For example, Q_2, with priority 5, is more important than Q_3, with priority 10. Throughout the chapter, the terminology used is that "Q_2 has a higher priority than Q_3", even though their configured priority values are inversely proportional, as explained. Albeit Q_2 receives all the extra available bandwidth, Q_3 would still receive additional bandwidth when Q_2 would have received its maximum allowed as configured through R_{max}. However, since for this test case the maximum rate for Q_2 is equal to the port capacity, Q_3 would receive additional extra bandwidth only if the traffic flows in Q_2 would lower the rate under 2.6 Mbps.

3.1.7 Proposed improvements for the prototype

OF has evolved significantly over the years, with many features being added to recent versions of the protocol. The QoS mechanism described in this chapter can be implemented by using any version of the OF protocol. OF defines a set of instructions for directing traffic flows to a specific queue on the output port [22]. These instructions are useful for realizing the QoS control mechanism proposed in this chapter. Two possible improvements for the implementation are discussed in this section: the first one relies on the OF pipe-line processing using multiple tables, which was first introduced in OF v1.1 [23], and the second improvement is based on the *Meter* table introduced in OF v1.3.0 [24].

Improvement 1

Because per-VC control is required, the number of entries in the flow tables in the data plane devices is increasing linearly with the number of VCs, rapidly exhausting the memory of the devices. This is also known as *flow table explosion* [4], and it is a well-known challenge in SDN, with several solutions being proposed in the literature [15] [9] [17]. By leveraging the pipe-line processing using multiple OF tables (OF v1.1), *operational model B* can be implemented efficiently with respect to the number of OF entries in the flow tables. In *Model B* (3.1.5), several VCs may share the same path in the data plane and differ only by their traffic class (i.e. output queue). Therefore, at the ingress forwarding device, the packets belonging to a VC are marked with a VC Identifier VCID (denoting the path) and a QoS ID (denoting the output queue assigned to that VC). The implementation of the VCID and the QoS ID may be VLAN tagging, MPLS labeling, etc. Under this model, in the provider core network, the forwarding and the enforcing of the QoS control mechanism is done solely based on the VCID and the QoS ID: the VCID dictates the output port for the packets, and the QoS ID dictates the output priority queue. In OF v1.0, this can be implemented by using a

single OF flow table as illustrated in upper part of Figure 3.10. However, this behavior can also be implemented with two flow tables (in OF v1.1), as illustrated in lower part of Figure 3.10:

- Table 1 contains the entries corresponding to the VCIDs so that that a packet is matched based on the VCID, and an ”output to port” action is added to the action list if a match occurs. The packet is then sent to the second flow table for further processing.
- Table 2 contains the entries for the QoS IDs so that the packet is matched against its QoS ID and an ”enqueue” action is added to the action list. At the end the actions in the action list will be applied to the packet.

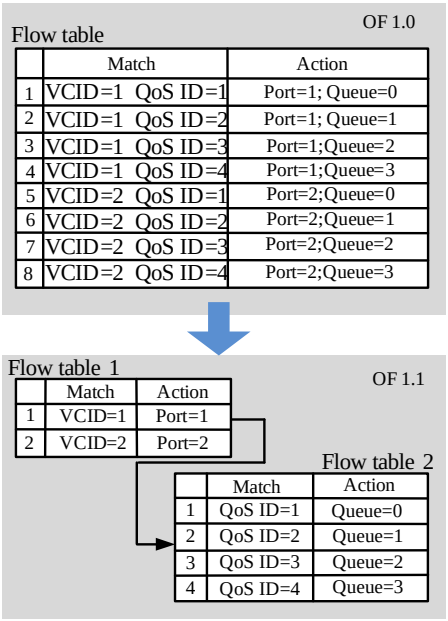


Figure 3.10: Comparison between the OF 1.0 and the proposed OF.1.1(or newer) implementation

This solution saves flow table space inside the forwarding devices when compared to an OF 1.0 implementation (Figure 3.10). This is especially valid if multiple traffic classes are used (each with its own priority queue), and several VCs are mapped on the same end-to-end path, because they share the VCID. In this case, the reduction in the number of flow entries can be even higher than for the simple case depicted in the figure.

Improvement 2

The OF v1.3.0 specification [24] introduces the concept of *Meter* table, which contains meter entries that can be used to implement various QoS mechanisms. A meter (i.e. an entry in the Meter table) can be associated with an OF flow entry in one of the flow tables. The meter is then used to measure the rate of the traffic flows matching the associated OF entry. A specific action is applied to each packet of a traffic flow, when the measured rate reaches a certain level. Two meter actions are included in the OF v1.3.0 specification: *drop* and *DSCP remark*. By applying the *drop* action, it is possible to implement per-flow rate limiters: the packets are dropped when the measured rate is above a threshold specified for that meter. The possibility to create per-flow rate limiters by using OF, eliminates the need to create the rate limiting queues through the OVSDB protocol, which is a more complex operation as it will be showed in the next section. This simplifies the implementation and reduces the initial flow setup delays. The *DSCP remark* can be used to create policing functions that decrease the priority of a traffic flow when it reaches a certain rate.

3.1.8 Summary

This section proposed an SDN solution for QoS aware service provisioning by combining a prioritization based QoS mechanism with a CBR algorithm. This resembles the traditional DiffServ-enabled MPLS-TE technique, however, being implemented in a centralized SDN architecture. The current work contributes to the field by discussing the aspects of QoS aware network resource provisioning at every layer of the SDN architecture. A detailed data plane QoS model is formulated, for which a corresponding control plane architecture is designed. The data plane QoS mechanism has been tested and validated by using OVS software switches, and OF and OVSDB as protocols on the D-CPI. In SDN, the resource management relies on manipulating the logical resources inside the SDNC. Through correct resource management, and by keeping the invariants presented in section 3.1.3, congestion in the network can be controlled, leading to predictable QoS. Two operational models for QoS service provisioning have been described in this work, which are realizable with existing technology, and cover various real world scenarios for service provisioning.

Based on the understanding obtained through this work, several areas for further research can be defined. One possible investigation can focus on fully implementing the two models for service provisioning and thoroughly comparing them with respect to the maximum achieved throughput, QoS isolation between VCs, request blocking rate, etc. However, since the core problem of this thesis is to ease QoS management through SDN, the next section will continue with investigating solutions for simple QoS configuration.

3.2 APIs for QoS configuration in SDN

Openflow (OF) is the most used protocol in SDN on the D-CPI interface, allowing the SDNC to control the forwarding of the traffic flows in the data plane. However, OF alone is not enough to support the more complex SDN services that need low level data plane control and management (i.e. configuration). As exemplified in the previous section, for low level configuration of network resources (e.g. priority queues) OVSDB has been used to complement OF. In consequence, there are two important pillars for QoS provisioning with respect to network performance, *configuration of resources* and *traffic forwarding*, and they often require two complementary protocols.

A key aspect of the SDN paradigm is to make networks programmable. This programmability further enables automation of various control and management functions such as provisioning of services, network recovery, etc. Seen from a holistic perspective, these aspects influence the QoS for the network services. For example, customers can expect a smaller provisioning time, less downtime due to fast and automated recovery, etc. To materialize the vision of having programmable networks, it is necessary to create APIs for managing and controlling the network. These APIs enable the interaction between external entities (e.g. services, administrator, dashboards, etc.) and the SDNC, which comprises the control logic of the network. As a result, the capabilities exposed by the SDNC at the A-CPI, where the northbound APIs exist, directly influence the network services and functions built on top of the SDNC. A rich set of APIs can support complex and compelling services with predictable QoS, while fewer or poorly designed APIs may hinder the applicability of SDN and the adoption of QoS mechanisms. To exemplify, some services and applications need a lower level API for QoS control (e.g. traffic engineering that relies on device specific QoS configuration), while other need a high level API (e.g. multimedia service requesting connectivity with QoS). In this context, it is important for the SDNC to provide APIs for QoS service provisioning at all the granularities. By doing this, a large variety of external entities (e.g. services, applications) can deploy QoS by interacting with the SDNC.

This section describes the design of SDN APIs for QoS configuration, continuing with the architecture illustrated in the previous section, but focusing on the details for data plane QoS configuration. Specifically, the API proposed here, termed *QoS Config API*, allows SDN services to configure priority queues on the ports of the data plane devices. As suggested in the previous section, the QoS configuration mechanism deployed in this work is using the OVSDB protocol. Therefore, the work presented in this chapter is aimed towards SDN deployments that are based on OVS because it inherently supports OVSDB. The *QoS Config API* is designed to be consumed by SDN services that need low level QoS control over the data plane through priority queue configuration, mechanisms that were described in section 3.1. Such SDN services (e.g. VC provisioning) use the QoS Config API to realize their logic, and further provide higher level network services to cloud or enterprise applications that need network

support with predictable QoS.

This section contributes to the research community with presenting the design and performance of a novel architecture for QoS configuration in communication networks, based on SDN principles. Several challenges that arise for the proposed architecture are addressed. The focus of the work presented in this section is to enable simplified QoS management by leveraging the SDN programmability thus contributing to the overall challenge of making QoS provisioning more appealing to NSPs.

3.2.1 Related Work

SDN is applied to offer bandwidth on demand between data centers in [15]. This service is evaluated with respect to the utilization of network resources, and the bandwidth allocation fairness between the traffic classes. When compared to traditional MPLS-TE technologies, their solution shows significant improvements. Like the work described in this chapter, in [15], the data plane QoS is based on priority queuing.

Another area of research in SDN is devoted to the design of APIs exposed by the SDNC towards services and application at the A-CPI. In [19], an SDNC architecture is proposed, which exposes an API for QoS aware service provisioning based on OF. However, the authors do not present the precise details of the API, thus limiting the understanding of the proposed API abstractions and their applicability. The solutions described in [5] and [11] focus on policy based QoS enforcement by using the features provided by OF. These policies are aimed at applications requesting network connectivity, or administrators managing the network.

Even though there are several works presenting high level APIs based solely on OF, there is little research and very few proposed solutions for QoS configuration. To the best of author's knowledge, the only work describing a similar solution is [28]. However, [28] focuses on the implementation details, while this section deals extensively with the challenges of the architecture, the design of the QoS Config API, and demonstrates its applicability and performance.

3.2.2 Architecture

The architecture used in this section is depicted in Figure 3.11. This architecture is very similar to the one described in section 3.1, however, it is modified according to the work presented in this section. Overall, the architecture follows the same layered design, with the data plane being based on OVS (through Mininet), and the control plane on Floodlight. There is also the applications layer, comprising a sample client application (termed simply *Client* or *Client application*), which consumes the API exposed by the SDNC.

To emulate the data plane, several OVS switches are deployed in a Linux machine. All OVS instances fetch their configuration from a common database residing on the same machine. This database is named OVS database (OVS Db), and it can be modified by using the OVSDB protocol, allowing remote entities to configure the state

of all the OVS instances inside the machine. An example that is of interest to the work presented here is priority queue configuration. To ease the understanding, the database is not depicted in the figure below, hence, the SDNC is shown to directly configure the OVS instances.

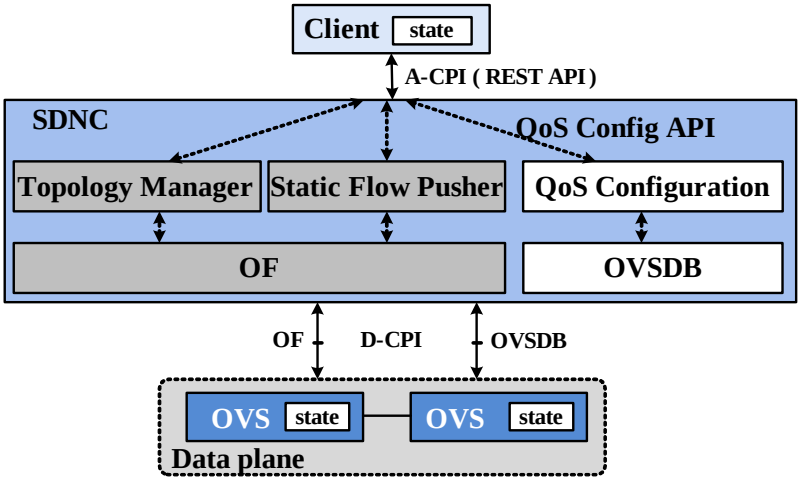


Figure 3.11: SDN architecture for QoS configuration

The grey SDNC modules in the figure are reused from Floodlight while the *OVSDB* and the *QoS Configuration* modules have been added for this work. As mentioned in section 3.1, *OF* and *OVSDB* modules realize the communication with the data plane devices through the corresponding protocols, OF and OVSDB. Apart from OF, two other modules have been reused from Floodlight. The first one, *Topology Manager*, maintains a logical topology representation of the physical network infrastructure. Besides, it offers a REST API for querying the state of the topology (e.g. switches, links, etc.). The second module, *Static Flow Pusher*, uses the features exposed by the *OF* module to provide a REST API for installing OF entries in the forwarding devices.

When implementing the OVSDB module, parts of it have been reused from the OpenDaylight OVSDB open source project [27] (i.e. the library for parsing the protocol messages). On top of OVSDB, the QoS Configuration module enables the *QoS Config API*, which represents the core of the work presented in this section. In particular, the QoS Configuration module provides methods to configure the priority queues in the OVS instances. For this, only a subset of the features exposed by OVSDB are used, given that OVSDB has a more generic purpose than just QoS configuration. There are two approaches to consuming the *QoS Config API*: as a Java API - inside the SDNC, or as a REST API from external entities.

To demonstrate the possibility of creating applications and higher level network

services based on the *QoS Config API*, a *Client* application is also added to the architecture (Figure 3.11). As illustrated in the figure, the configuration state for the priority queues is kept in the data plane and in the *Client* application, indicating that the QoS Configuration module is a stateless entity.

3.2.3 QoS Configuration Module

Configuration of OVS instances must be performed in accordance with a specific data model, which resides in the OVS Db. Since this data model is often too complex to be useful to higher layer SDN services, the purpose of the QoS Configuration module is to abstract away this complexity and expose a simpler, thus easier to use data model, which is accessible through the *QoS Config API*.

Data plane abstraction

There are three data models, at every layer in the proposed architecture (Figure 3.12). On the left side of the figure there is the *OVS data model*, which is represented as tables inside the OVS Db [25]. This database, and implicitly the data model, is accessed by the SDNC through the OVSDB protocol. In this model, a switch is denoted as a *Node*. Further, within each OVS *Node*, several *Port* objects may exist, denoting a real port on an OVS instance. A *QoS* object can be added on a *Port* in order to hold QoS related configuration state that belongs to a *Port*, such as the maximum rate for the port. This rate is configured here to be the capacity of the outgoing link. At the leafs of the OVS data model, there are the *Queue* objects, which represent the target entities for configuring QoS in the network. As described in [25] and also presented in section 3.1, the parameters that can be configured on the queues are: R_{min}, R_{max}, *size* and *priority*. These parameters have the same semantics as explained in section 3.1, allowing for complex QoS mechanisms (e.g. similar to Diffserv) to be implemented in the network [1].

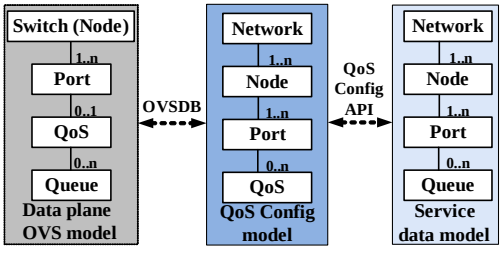


Figure 3.12: Data models available at each layer of the proposed architecture

On the right side of the figure, there is the service data model, which is made available through *QoS Config API*. As the data model indicates, a service (or a client

application) has access to priority queues (i.e. *Queue* object) that are associated with a certain port, switch, or with the entire network. This is made possible through the QoS Configuration module, which maps the OVS data model to the data model that services can use through the *QoS Config API*. For this, the QoS Configuration module facilitates the mapping between the two data models, by keeping state about the topology and the *QoS* objects configured for each port in the network. This mapping is suggested in Figure 3.12 through the data model in the middle (i.e. QoS Config model).

State distribution

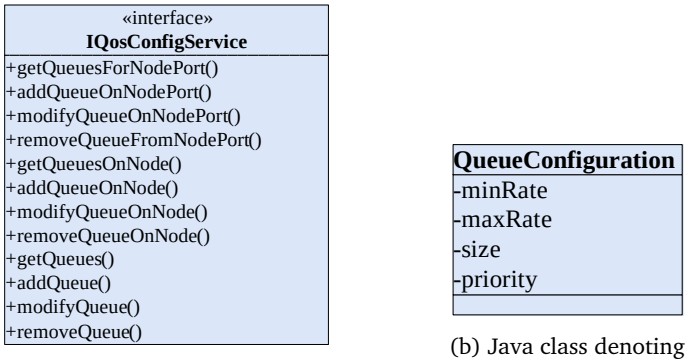
In order to have a dynamic QoS aware service, the state of the priority queues changes very often throughout time. Keeping the configuration state of the priority queues inside the QoS Configuration module would require complex mechanisms to maintain this state consistent with the state in the OVS database (at the data plane), due to the frequent configuration changes. Alternatively, the state of the *QoS* objects does not change over time, and thus keeping the *QoS* objects in the QoS Configuration module does not require state consistency mechanisms to be employed. As a result, the QoS Configuration module does not keep state about the priority queue configuration (Figure 3.12), but it keeps state about the *QoS* objects associated with each port in the network. With this mapping, it is easy to access the priority queues in the OVS data model, which are stored in the OVS Db.

This decision regarding state distribution at every layer in the architecture is a compromise between the performance of the *QoS Config API* and the implementation complexity. On one hand, the performance is affected because the queue configuration state resides only in the data plane (in the OVS Db), hence, in order to access this configuration state, several queries are initiated towards the data plane from the QoS Configuration module. On the other hand, this stateless implementation is less complex than a stateful one (which requires state consistency mechanisms). Additionally, the stateless implementation presented here allows to easily extend the QoS Config API with new parameters, without major enhancements or further performance implications.

QoS Config API design

Figure 3.13a shows the Java methods comprised by the *QoS Config API*. These methods are also available as REST APIs. As it can be noticed, the queues can be configured with three granularities. At the lowest level, queues can be configured on each individual port in the network. A port in the network is defined as a *NodePort* object in Figure 3.13a, since the node and port association uniquely identifies a certain port. At the next level, a queue can be configured on all ports of a given switch (i.e. *Node*). Finally, a queue configuration can be pushed to all ports in the network (highest level of

granularity). These different granularities ease the implementation of a service since it allows the service to focus on managing several queues at once, grouped by the topological element they belong to (e.g. device, network).



(a) Java API methods

(b) Java class denoting a queue configuration

Figure 3.13: Java API details for the QoS Config API

Each priority queue has an associated number which is unique for the containing port. This number is passed as argument to the API method calls in order to identify a certain queue. Moreover, every method of the *QoS Config API* takes as argument the topological element for which the queue must be configured. Specifically, a node is identified by a Datapath Identifier (DPID) and the port by a port number. Some API methods take as argument also a queue configuration, which is illustrated as a Java class in Figure 3.13b. There are four operations that can be executed when configuring a priority queue:

1. *Get*: returns all the queues configured on a certain topological element (e.g. port), together with their configuration.
2. *Add*: a queue, with a given configuration, is added on every port that the method refers to. For example, one queue is added on every port in a certain device.
3. *Modify*: all the queues with a certain number, belonging to the topological element indicated, are modified to the queue configuration passed as an argument to the method.
4. *Remove*: removes a certain queue.

3.2.4 Example of client SDN service

An example of a simple application that consumes the *QoS Config API* and provides a more complex service is illustrated in Figure 3.14. In this example, the *Client*

application offers a service termed *QoS slice*. A *QoS slice* is a set of priority queues that are configured in the network and are treated as a single unit. By configuring priority queues, a QoS slice comprises a portion of the capacity of each link in the network, denoted by the serving rate for the priority queues on the output ports. To implement a QoS slice, the *Client* application maintains state about the configured set of queues. Trough the REST API exposed by the SDNC, the *Client* application has access to the entire network topology, together with QoS configuration capabilities for the devices in the network. For this simple example, all the network devices are considered when defining a QoS slice, thus, the topology of the slice always resembles the real network topology. It is possible, however, to enhance the simple *Client* application to create *network slices*. A *network slice* would then include a subset of the network devices, together with the priority queues on each port in the devices. Moreover, if combined with traffic isolation mechanisms, a *network slice* can become a Virtual Private Network (VPN).

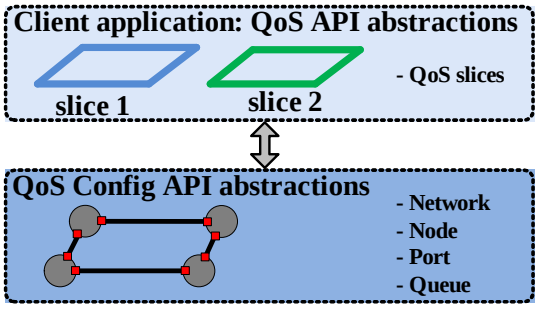


Figure 3.14: Example of client application based on the QoS Config API

This client application example is used to qualitatively demonstrate the capabilities of the *QoS Config API*. The specific testbed configuration and results for this qualitative assessment are described in the next section.

3.2.5 Tests and Results

The prototype implementation is tested on an OVS based distributed testbed. Unlike section 3.1 where the testbed is based on Mininet, for this section the Distributed Openflow Testbed (DOT) [32] is used to emulate the data plane. Two physical servers host the complete testbed (Figure 3.15). The first server (i.e. Server 1) hosts two VMs: in VM1 the *Client* application is executed and VM2 contains the SDNC. The *Client* application creates the QoS slices as described in the previous section. Another purpose of the *Client* application is to manage the test lifecycle: define and execute tests by coordinating the traffic generators in the network. The second server (i.e. Server 2) hosts the data plane comprising four OVS instances (sw1 to sw4) and

three VMs representing the network hosts (h1 to h3). Each VM contains a traffic generator and is attached to one of the OVS switches in the network. A different traffic generator, D-ITG [7], has been used in this section to obtain more accurate throughput measurements than in the previous section. Moreover, unlike in section 3.1, here, the throughput is measured directly at the output ports of the network devices by using OF *Statistics Request* messages. By measuring the data rate directly at the port, the results are more accurate than the results reported by the traffic generator which does not include Ethernet protocol headers.

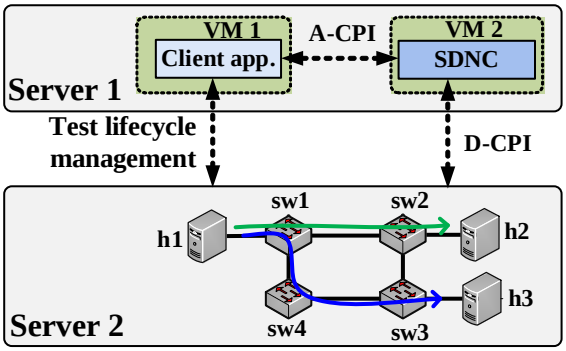


Figure 3.15: Testbed setup

To demonstrate the capabilities of the QoS Config API and assess its performance, two tests have been performed.

Test 1

The first test is qualitative and it demonstrates the capabilities of the *QoS Config API* by using the QoS slicing application. From this application, two QoS slices are created (also depicted in Figure 3.14), each slice consisting of a set of priority queues, one queue per output port in the network. A bandwidth of 7 Mbps is allocated to *Slice 1* and 3 Mbps to *Slice 2*, by configuring the priority queues on all the ports. Next, two traffic flows (green and blue) are generated in the network, as illustrated in Figure 3.15. Each flow has a rate of 10 Mbps, which is enough to saturate any of the two QoS slices. The green flow (*h1* to *h2*) belongs to *Slice 1*, and the blue flow (*h1* to *h3*) belongs to *Slice 2*. The traffic flows are forwarded to the priority queues according to the slice they belong to, by installing OF rules in the network devices through the REST API of the SDNC. The duration of the test is 100 s.

Three network links are monitored in order to understand how bandwidth allocation changes when controlling the slice configuration from the *Client* application. The link between *sw1* and *sw2* contains the green flow, which characterizes the bandwidth

allocation for Slice 1. Links *sw1-sw4* and *sw4-sw3* transport the blue flow, which characterizes Slice 2. Since each of the links carries traffic only from one slice (*Slice 1* or *Slice 2*), by monitoring the entire link bandwidth utilization, it is possible to infer the bandwidth allocation per slice. This setup allows for precise measurements, which are performed by reading the counters at the output port.

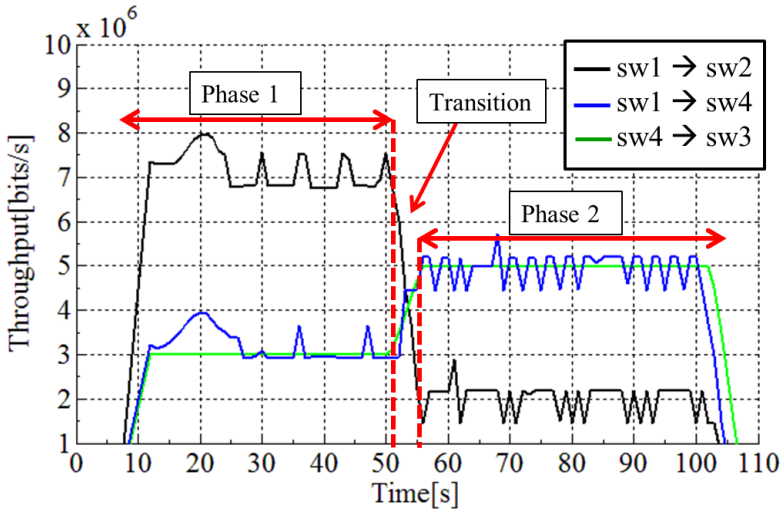


Figure 3.16: Qualitative results for the QoS Config API

There are two phases during the test, each lasting for approximately 50 seconds (Figure 3.16). As the results show, in the first phase (i.e. Phase 1), the measured throughput conforms with the bandwidth configuration for each slice (i.e. 7 and 3 Mbps respectively). At the end of the first phase, the *Client* application modifies the configuration of the QoS slices by using the QoS Config REST API. Specifically, *Slice 1* is reduced to 2 Mbps and *Slice 2* is increased to 5 Mbps. This new change in the QoS configuration for the two slices denotes the second phase of the test. As it can be noticed, during the second phase, the monitored throughput conforms with the new configuration. Between the two test phases there is a short transition period, when the change in the priority queue configuration is taking effect. Even though the new configuration is enforced very fast, the transition period takes several seconds. The reason for this is the delay in monitoring the traffic. Moreover, the throughput is continuously averaged over five measurement samples in order to eliminate instantaneous spikes in the monitored data. This also makes the transition period to be perceived as longer than what the real transition takes in the forwarding devices.

The results for the first test demonstrate that the *QoS Config API* can be used to

implement QoS aware network services and also to dynamically control the allocated bandwidth at every port in the network. It is important to notice that provisioning of QoS is greatly simplified through programmability and by designing APIs with the correct level of granularity according to the target application. This is demonstrated by the toy example (i.e. QoS slicing application) used in this section.

Test 2

The purpose of this test is to assess the performance of the QoS Config Java API. In this case, the performance is denoted by the *response time of the API calls*. Only the Java API is tested in here since it better reflects the performance of the SDNC modules, by eliminating the additional impact of making REST API calls. Four API methods have been selected for testing, out of the twelve available (Figure 3.13a). These are the first four methods illustrated in the Figure 3.13a, which refer to individual ports.

To evaluate the response time, requests are initiated through the QoS Config Java API, and the total Round Trip Time (RTT) to successfully complete the request is measured. This RTT includes processing time inside the QoS Configuration and OVSDB modules, and the RTT of the OVSDB messages on the D-CPI interface (OVSDB Control Channel). The RTT on the OVSDB control channel is measured and represented also individually, to better quantify the individual time contributions of each component in the architecture. In order to have a statistical proof for the results, 500 executions have been performed for each API call.

Figure 3.17 shows the test results. It takes approximately 7 - 8 ms to complete a API request for the *Add*, *Modify* and *Remove* methods. However, the *Get* method takes 14 ms to complete. The difference in RTT for the *Get* method is due to slower processing inside the SDNC modules, since, this request carries more data which must be parsed. In all the cases, the average RTT for the OVSDB channel (at the D-CPI) is 2 ms (Figure 3.17).

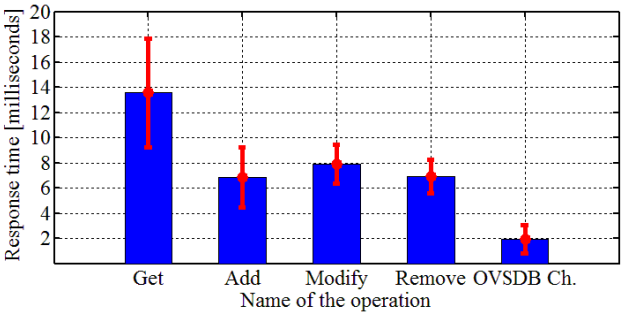


Figure 3.17: RTT for the API calls and the OVSDB control channel

The results show that the *QoS Config API* has a good performance. As a result, The proposed API can be used to control the QoS configuration in the forwarding devices dynamically, in real time. For the case when the SDNC has a reactive behavior (reactive OF entries installation), it is still possible to perform QoS configurations per flow, since the flow setup is delayed only with 7 ms. Overall, the results show that the *QoS Config API* performs better than similar APIs reported in the literature [28].

3.2.6 Discussion

Two testbeds have been deployed and used in this chapter to test the proposed ideas. It is important to note here a short comparison between the two, given that this comparison is a basis for future decisions regarding the OF testbeds used in this thesis. Mininet is a collection of APIs in Python, which uses lightweight Linux virtualization to create the network testbed. Each host is a user namespace and each OF forwarding device is an OVS instance. The entire Mininet emulated testbed is deployed in a single machine. This machine must support not only the testbed itself but also traffic generation for the emulated test network. Due to performance reasons for executing everything in a single machine, other alternatives have been considered. Such an alternative is DOT, which has been used in section 3.2. DOT relies also on OVS instances to emulate the OF forwarding devices. Additionally, DOT uses VMs as network hosts, which are much more heavyweight when compared to the hosts used in Mininet. Specifically, VMs take more time to create, deploy and configure, and also consume significantly more resources. DOT was considered promising since it was built to support distributed experiments, thus, allowing to use more machines and overcome potential performance issues. After using both DOT and Mininet, it was clear that the benefits brought by DOT with respect to testbed distribution and performance isolation were surpassed by the difficulty in deploying and maintaining it. Moreover, it has been proven later during the research work that the performance of some traffic generation tools (i.e. Iperf3) has improved, allowing for a large amount of traffic generated in the network, with a small performance impact. For these reasons, Mininet continues to be used for the remaining work in this thesis.

3.2.7 Summary

This section extended the architecture and concepts introduced in section 3.1 by presenting an API for QoS configuration in SDN. Together with the features provided by OF, the proposed *QoS Config API* opens the possibility to implement SDN based QoS aware network services. This has been emphasized through the simple QoS slicing client application proposed herein. The capabilities of the *QoS Config API* have been demonstrated and its performance allows for dynamic, fast (under 10 ms) configurations.

Further performance improvements can be made for the *QoS Config API*. The RTT on the OVSDB control channel cannot be improved since it depends on the capacity

and the latency of the control channel. It is possible to reduce the number of OVSDB requests by multiplexing several operations in the same OVSDB message. This is useful in cases where multiple queue configurations need to be executed simultaneously.

For this work, the initial OF version (i.e. 1.0) has been used. This version offers a limited set of QoS features. More specifically, it enables forwarding of traffic flows to a certain queue inside the network devices. However, the queue configuration must be performed through a different protocol (e.g. OVSDB). In more recent OF versions (e.g. 1.3.0), new QoS capabilities have been added such as *Meter* tables. With this new feature, it becomes possible to use OF to rate limit traffic flows or to mark them with a specific DSCP value. Hence the need to deploy additional management protocols is reduced.

3.3 Chapter Summary

While network overprovisioning can offer an immediate solution to poor service quality in communication networks, it is not a sustainable solution in the long term, as argued in chapter 1. Proper QoS aware service provisioning is the alternative to network overprovisioning. To support the adoption of proper QoS mechanisms in network infrastructures, new technologies must focus not only on providing services with QoS, but also on simplifying the management of such QoS mechanisms. SDN is a novel technology which brings several benefits, allowing for efficient resources utilization and simplification of network management through programmability and automation. This chapter argues that SDN is particularly suitable to enable proper QoS provisioning in communication networks. To this end, a complete framework for QoS aware service provisioning is proposed in section 3.1. This framework is designed to fit a variety of network service models, where the Virtual Circuit is the fundamental unit for service offering. Further, the framework is validated in an emulated SDN testbed.

As it emerged from section 3.1, there are two important pillars for QoS service provisioning in SDN. Specifically, the first pillar is traffic control through OF, and the second pillar is data plane resources provisioning and configuration. Section 3.2 investigates the mechanisms that comprise the second of these two pillars, by proposing a QoS configuration API. This API is designed to be simple and versatile, thus, facilitating the adoption of QoS techniques in communication networks. Additionally, the capabilities of the API are demonstrated and its performance assessed. The results show that the API allows to easily create more advanced and innovative SDN services that require QoS from the network (i.e. QoS slicing). Moreover, the results prove that the performance of the QoS Config API is good enough to support real time priority queue configuration.

CHAPTER 4

Data plane challenges for QoS provisioning in SDN

The previous chapter described a complete architecture for provisioning of network services based on the SDN paradigm. A key point regarding the architecture is the focus on delivering predictable QoS for the network services, by careful configuration of network performance parameters. Apart from the network performance parameters which have been considered in Chapter 3 (e.g. bandwidth, delay), there are other aspects that influence the perceived QoS for a network service. For example the service provisioning time, the service downtime, etc., can influence the level of quality for a certain service, as perceived by the customer. Moreover, since the services provisioning is realized by using the SDN paradigm, challenges specific to the SDN architecture arise. Starting from this chapter, the thesis focuses on identifying the most important challenges in the SDN architecture that could potentially impede QoS provisioning in large network domains. This investigation is performed separately, for each layer of the SDN architecture, starting from the data plane in this chapter, and continuing with the upper layers in the following chapters. The work presented in this chapter is based on Paper D.

The data plane in the SDN architecture contains the network resources comprising the forwarding devices and the links between these devices. For QoS aware service provisioning, these network resources must be configured accordingly, as described in the previous chapter. When some of the resources become unavailable, provisioning of QoS is hindered. Hence, it is critical to maximize the utilization of the available data plane resources in order to facilitate QoS in the SDN architecture.

In an OF based network, each forwarding device matches incoming data packets to OF entries in the flow table. For OF v1.0, if there is no match for a new packet, the packet is sent to the SDNC for further processing, while for newer versions of the protocol (e.g. v1.3) the packet is dropped by default. Since the correct packet processing relies on having OF entries installed in the flow table of the devices, the flow tables represent an important network resource. If this resource becomes scarce, the normal provisioning of network services is interrupted, as a result services must wait for the resource to become available or traffic must be routed through devices that have space in their flow tables for new OF rules.

A common way to implement the flow tables in OF devices is by using the Ternary

Content Addressable Memory (TCAM) [2]. This special type of memory allows flexible and fast matching of OF rules against packet headers. Moreover, it also enables wildcard matching (i.e. don't care bits), meaning that parts of the packets headers are not considered for the matching. Although TCAM has many benefits, this type of memory is also very expensive and power demanding thus it cannot be used in excess in network devices. For this reason, OF forwarding devices have limited resources (i.e. TCAM) to implement the flow tables. This limits the number of OF entries that can be installed in a device, making the flow tables a scarce resource in the network. Thus, intelligent mechanisms are needed to increase the utilization of the flow tables.

This chapter describes a solution to efficiently utilize the space in the flow tables in OF devices. The proposed solution leverages the idea that the number of OF entries used to forward a certain amount of traffic can be reduced by aggregating those traffic flows. By doing this (i.e. multiplexing), several traffic flows can be forwarded as a single unit (termed *aggregate* in here), requiring fewer OF rules. Ideally, traffic flow aggregation should occur at the edge of the network, so that the core of the network forwards only traffic *aggregates*. In this case, the number of OF entries is reduced in the core of the network, as compared to forwarding all the individual traffic flows. However, instead of having fixed aggregation points in the network, the proposed solution relies on a mechanism which dynamically chooses when and where to apply aggregation. Moreover, the mechanism considers the impact of the aggregation on the QoS of the traffic flows.

The efficiency of the solution proposed here is defined by the reduction in the number of OF rules achieved in the network by aggregating the traffic flows, as compared to forwarding the individual traffic flows. A factor that impacts the efficiency of the proposed solution is the number of flows that are multiplexed into an aggregate, which is further influenced by the traffic patterns and the network topology. In this sense, several tests have been performed for different topologies, to better understand the impact of the topology upon the aggregation mechanism.

The remaining of the chapter is organized as follows. Section 4.1 presents background information in relation to the proposed solution, also referring to existing research work. Section 4.2 describes the implementation of the aggregation service prototype, following that section 4.3 shows the results for the tests performed with the implemented prototype on various topologies.The chapter is summarized in section 4.5.

4.1 Background and related work

4.1.1 OF based SDN architecture and principles

OF forwarding devices use the concept of *traffic flow* to process the traffic in the network. A traffic flow is a stream of packets uniquely identified by a set of packet header values. The first packet of a traffic flows has a particular significance, especially

for reactive OF controllers, since it is usually sent to the controller in order to trigger OF rule installation in the flow tables of the forwarding devices. After the OF rules are installed, the subsequent packets belonging to that traffic flow have already a valid match in the flow table and are processed accordingly. In such a reactive operational mode (i.e. SDNC reacts to new traffic flows and installs OF rules), the SDNC is always aware of the traffic flows that are forwarded in the network at a certain time.

As previously mentioned, TCAM is a common technology used to implement the flow tables in the forwarding devices, because it supports flexible wildcarding for the packet headers, and high speed matching against the packet headers (i.e. one memory access for the entire flow table) [2] [10]. The downsides of TCAM are the high cost and high power consumption, leading to increased costs for building and deploying forwarding devices with large flow tables [9]. Limited space in the flow tables impedes the adoption of OF based SDN for environments that have a large number of traffic flows. For example, current commercial OF devices support a few thousands OF entries, which is insufficient even for smaller environments [2] [9] [11]. To emphasize, the limited amount of OF entries impacts the network service provisioning, which relies on traffic forwarding, further affecting the QoS of the network services offered to the customers.

4.1.2 Overview of Aggregation Service

To cope with the challenges outlined above, the number of OF entries in the forwarding devices must be reduced without affecting the existing functionality in the network. A possible method to achieve this objective is to treat several traffic flows as a single *aggregate*. In this way, multiple traffic flows are forwarded within a device by using a single OF rule, instead of several fine grained OF rules. This operation is termed *traffic flow aggregation* here. Based on traffic flow aggregation, a novel service (i.e. Aggregation service) is proposed in this chapter which dynamically aggregates traffic flows in the network by taking into account the computed routes and the QoS demands of the flows.

To be able to treat a set of traffic flows as a single *aggregate*, the SDNC must use an identifier for these traffic flows, which then are marked with the identifier and forwarded in the network based on their marking. There are various identification mechanisms that can be used such as VLAN tagging, MPLS labeling, etc. While these mechanisms are commonly used in traditional networks, they are also suitable for the aggregation service proposed here.

As mentioned, the efficiency of the proposed solution is denoted by the reduction in the number of OF entries in the forwarding devices, with and without the aggregation service. Thus, the choice of the identification mechanism is not relevant with respect to assessing the performance of the service. Specifically, the number of OF rules in the forwarding devices does not depend on whether VLAN tagging or MPLS labeling is used to identify aggregates. Based on this, the identification mechanism used here

is VLAN tagging, which is both simple and commonly used in SDN implementations. The limitations imposed by VLAN on the number of tags (approx. 4000) does not impact the current work. However, in a real deployment, the choice of encapsulation technology is very important and usually depends on several factors such as hardware support, encapsulation overhead, number of available identifiers, etc.

The proposed aggregation service works autonomously to reduce the number of OF entries in the devices. Specifically, the service listens for *Packet In* notifications and decides whether the new traffic flows, denoted by the *Packet Ins*, can form new aggregates with existing traffic flows, or be included in existing aggregates. In taking this decision, the service uses two criteria, which can also be configured according to high level policies to make the service flexible to network changes:

- The degree of overlapping between the routes taken by the traffic flows. In the current work, the aggregation service relies on the routes computed by the route computation service. In this case, aggregation is possible only if there is an overlapping between the routes of the considered traffic flows. This criteria is generically termed *path overlapping* here. Another possibility would be that the aggregation service actively intervenes by rerouting traffic flows to achieve a higher degree of overlapping. Nevertheless, this would require a significantly more complex aggregation mechanism. For example, in cases where the computed routes are bound to certain QoS requirements, the aggregation service would have to reroute the traffic flows and still maintain the QoS requirements for the new routes. Thus, in this case, the aggregation service should also perform constraint based routing. For simplicity, in the current work, the aggregation service uses the already computed routes, without intervening with rerouting traffic flows in the network.

- Qos is the second criteria considered for the aggregation service. It is important that while aggregating traffic flows, the QoS of the flows is not neglected. For example, voice call traffic cannot be aggregated with file transfer traffic, even if the traffic flows have a high degree of path overlapping. To this end, the proposed service aggregates only traffic flows that belong to the same QoS class, by considering the DSCP marking for the data packets belonging to each traffic flow.

4.1.3 Related Work

There is significant work devoted to improving the utilization of flow tables in OF devices. In solving this challenge, various approaches are used, ranging from specific hardware implementations and memory configurations, to higher level techniques such as OF rules compression. As an example, the authors in [11] combine SRAM memory, which is cheaper, with TCAM, to achieve a trade-off. Further, they argue that some of the simpler packet processing (e.g. MAC based forwarding) can be done by

using tables implemented in SRAM while the more complex processing, particularly involving wildcards, can be done by using tables implemented in TCAM. Through this trade-off the need for TCAM is reduced in the forwarding devices.

In a similar fashion, [12] suggests a solution based on mixing SRAM and TCAM. On top of this mixt memory structure, an OF rules compression algorithm is applied. This algorithm combines several finer grained OF rules into fewer coarser grained rules. However, since the complete solution is studied in the context of server load balancing, the focus is not on assessing the degree of rules reduction in the flow tables.

[7] and [8] take a higher level approach by focusing on OF rules compression inside the flow tables. Their solutions dynamically restructures the OF rules and the wildcard matching in particular, to forward the same number of traffic flows but with fewer OF rules. The fundamental idea behind this work is that wildcards can have a common prefix, hence they can be aggregated based on that prefix. This idea resembles the IP address subnet aggregation from traditional IP routers.

[4] is the work most similar to this chapter, since the authors argue for a similar idea that dynamic flow aggregation is suitable to reduce the number of OF entries in core network OF devices. Moreover, the same criteria are used for aggregation decisions in [4] as in this chapter. However, while in here DSCP markings are used as QoS criteria to guide the aggregation, in [4], the QoS is inferred through transport port numbers, basically relying on traffic similarity rather than on specific QoS demands. For the aggregation service proposed in here, QoS is specified by another entity through DSCP marking, thus making the service simpler by separating the functionality between various entities. Alternatively, in [4], the aggregation service must also decide what QoS level certain traffic type must have, adding more complexity to the service logic. Adding to the comparison, the work in this chapter provides a quantitative analysis of the aggregation service for various network topologies, thus contributing with key knowledge about what topologies are better suited for such a solution. Conversely, [4] does not provide a quantitative analysis, focusing instead on illustrating the service behavior (i.e. qualitative).

4.1.4 Backhaul Networks

A *backhaul network* aggregates traffic from various access networks (mobile or fixed), and transports it towards the core network. Backhaul networks must accommodate an increasing number of traffic flows, which result from an increasing number of devices and users. Due to the large number of traffic flows, the limitations imposed by the space in the flow tables are a challenge in backhaul networks. Moreover, to satisfy users' need for quality, end-to-end QoS differentiation (i.e. access to core) is critical and must be considered when forwarding the traffic through the backhaul network. Due to this reason, the topologies chosen here to assess the service performance, are typical backhaul network topologies. Nonetheless, the proposed solution is intended to be applicable in any OF based SDN deployment, and these topologies are used here

just to illustrate their impact on the service performance and behavior.

The aggregation service is implemented at the IP layer of the TCP/IP stack, meaning that the service can process IP packets and identify traffic flows at the IP layer. However, mobile backhaul networks may have a different protocol stack depending on the underlying technology. As a particular example, mobile backhaul networks use the GPRS Tunneling Protocol (GTP) to transport the user data packets. Even though OF does not support GTP, studying the behavior of the aggregation service on backhaul topologies is still valid since the focus of this work is not on the specific implementations or underlying technologies, but rather on the service logic. For the same reason, VLAN was chosen as an aggregate identification mechanism, although any other mechanism would be suitable. Additionally, since SDN has the potential to simplify network and traffic management, there is research work which investigates the applicability of OF to the mobile backhaul networks[3], making the aggregation service proposed here directly applicable to such scenarios.

4.2 Prototype implementation

The aggregation service has been implemented as an extension to the open source controller Floodlight [5]. Figure 4.1 shows the prototype architecture, together with a data plane forwarding device. At the control plane, there are three modules that contain service logic. Two of the modules, *Forwarding* and *Topology Manager*, have been reused from Floodlight. *Forwarding* module manages the OF rules installation in all the devices along a certain route, for the traffic flows that need forwarding in the network. *Topology Manager* module contains a logical representation of the network topology, together with the entire set of routes to forward traffic in the network, computed using a hop based shortest path first algorithm. The *Aggregation* module contains the logic for the aggregation service. Apart from these modules, there are two logical data stores that keep information needed by the Aggregation module. These two data stores are the *Flows Database* (Flows Db) and the *Aggregates Database* (Agg Db in the figure). The former data store (i.e. Flows Db) is used to keep information about the traffic flows that are forwarded in the network, while the latter (i.e. Aggregates Db) is used to keep information about the aggregates in the network.

There are two modules in the SDNC that have forwarding related functionality, which may be conflicting. On one hand, the Forwarding module installs a set of OF rules to forward individual traffic flows in the network, according to the routes computed and kept by the Topology Manager. On the other hand, the Aggregation module also installs OF rules in the devices in order to construct and forward aggregates, which comprise several individual traffic flows. Due to this conflicting logic, when a new traffic flow arrives in the network, there must be a precise mechanism to decide what module installs the OF rules for the new traffic flow, to avoid incorrect behavior.

OF messages sent by the forwarding devices on the D-CPI are converted into OF

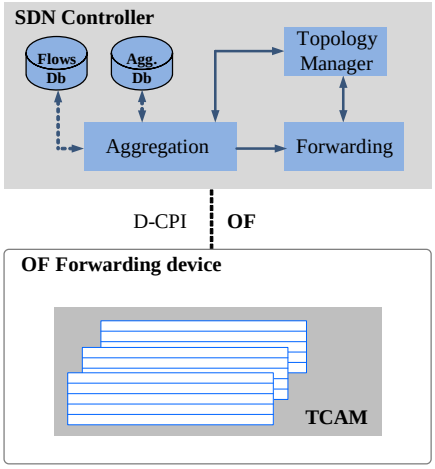


Figure 4.1: Generic SDN architecture for the aggregation service

events inside the SDNC. In Floodlight, each module can subscribe for OF events. The core Floodlight controller performs event dispatching in a predefined order to each of the subscribing modules, thus creating an *event processing chain*. With respect to this chain, the Aggregation module processes OF events before the Forwarding module. Moreover, after processing a particular OF event, the Aggregation module can stop the event processing chain so that the event is not dispatched to the Forwarding module. By leveraging this feature, the Aggregation module coordinates the entire OF related behavior to achieve a common and coherent logic for traffic forwarding.

4.2.1 Aggregation algorithm

There are two main operations that are performed by the Aggregation module. The first operation is to identify whether certain flows can form an aggregate or not. Depending on the degree of path overlapping, it may be the case that the flows can be aggregated only on a portion of their entire individual routes in the network. This portion where the traffic flows are aggregated is termed *aggregate segment* in here (Figure 4.2). With respect to an aggregate segment, there are three types of OF devices: ingress aggregating devices (S3 for green and red traffic flows), middle aggregating devices (S4), and egress aggregating devices (S5 for the red traffic flow). The second operation is to install the OF rules in these devices. Specifically, each aggregate is globally identified by a VLAN tag, hence, all the traffic flows to be aggregated must be tagged with the corresponding VLAN tag at the ingress aggregating device. Further, the traffic flows are identified and forwarded as an aggregate in the middle aggregating devices, and deaggregated at the egress aggregating device. To better

manage VLAN allocation, the Aggregation module maintains a pool with available VLAN tags. Whenever a new aggregate must be constructed, a free VLAN from the pool is selected, and when an aggregate is removed, the corresponding VLAN is released and added back to the VLAN pool.

Depending on the type of OF event, the Aggregation module executes different logic. In this sense, there are two types of OF events that trigger the execution of the aggregation algorithm, and they are discussed in detail in the following two sub sections.

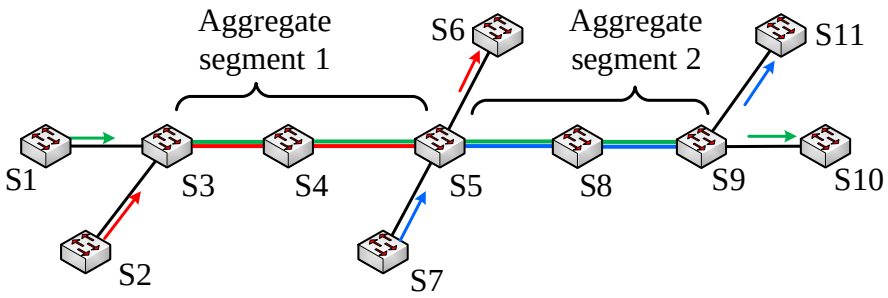


Figure 4.2: Aggregation segmentation

Arrival of a new traffic flow

When a new traffic flow arrives in the network, the first packet of the flow is sent to the SDNC thus generating a *Packet In* event in the controller. When this occurs, the logic depicted in Algorithm 1 is executed. First, the received packet is parsed and assigned to a local variable, *P* (line 1). Next, if *P* is not an IP packet then the processing of *P* is delegated to the Forwarding module. As mentioned, the aggregation service works at the IP layer hence it processes only IP packets. Alternatively, if *P* is an IP packet, then the aggregation algorithm updates the Flows Db with the new traffic flow denoted by *P*. Next, it extracts the route for *P* from the Topology Manager module, and it also extracts the DSCP marking from *P* (lines 5 - 6). The variable *Done*, which is initially set to *False*, marks the end of the aggregation procedure for the new traffic flow corresponding to *P*. In the **while** loop (line 9), the aggregation procedure is repeated until *Done* becomes *True*. First, the algorithm verifies if there can be an aggregation for the extracted *route* (line 10). This verification includes existing individual traffic flows in the network as well as existing aggregates, information which exists in the two data stores (i.e. Flows Db and Aggregates Db). If multiple aggregation possibilities exist, the algorithm choses the one with the highest path overlapping. If the route can be aggregated, the aggregation is performed and the corresponding OF rules are installed in the forwarding devices (line 11). After the aggregation is performed, the

Aggregates Db is updated with the new aggregate. Next, the *crop* procedure removes from the current route the segment which has just been aggregated, such that *route* takes the remaining non aggregated portion of the initial *route*. As illustrated in Figure 4.2, a route can be aggregated over several segments. Consider as an example the green flow in the figure, which was aggregated in two rounds: first on the S3-S4-S5 segment with the red flow, and then over the S5-S8-S9 segment with the blue flow. With the remaining *route*, the aggregation is triggered again. When a route cannot be aggregated, *Done* takes *True* (line 15). The aggregation procedure can be completed in two ways. First, the *route* can be *Null* meaning that it was aggregated over all its length, in which case the algorithm ends. Second, there may be a *route* that could not be aggregated (different from *Null*), which is either the initial *route*, or a route resulted from *cropping* the initial route due to successfully aggregating over a segment. In such cases, the remaining *route* is sent to the Forwarding module to be configured in the data plane as is, by using OF rules.

Algorithm 1 Aggregation algorithm: Packet In event

```
1:  P ← received packet
2:  if P != IP packet then
3:      Forwarding.process(P)
4:  else
5:      update flows Db
6:      route ← route forP
7:      dscp ← DSCP from P
8:      Done ← False
9:      while not Done do
10:         if canAggregate(route, dscp) then
11:             aggregate (route, dscp)
12:             update aggregates Db
13:             route ← crop(route)
14:         else
15:             Done ← True
16:             if route != Null then
17:                 Forwarding.install(route)
```

Arrival of a flow removal notification

All the OF rules have an idle timeout of five seconds hence if no data packet matches a rule for more than five seconds, the rule is removed from the flow table in the forwarding device. The OF rules installed by the SDNC in the data plane are configured so that the devices send an OF notification (i.e. OF flow removed message) to the SDNC when a rule is removed from the flow table. This is required in order for the

SDNC to have an up-to-date view of the existing traffic flows in the network. When a flow removal notification arrives at the SDNC, the Aggregation module process it and executes the logic described in Algorithm 2.

There are three types of OF rules that are being removed from the flow tables in the forwarding devices. The first type is represented by OF rules that correspond to aggregates. This type of OF rules are termed *aggregate rules* in here. An aggregate rule is removed when all the traffic flows belonging to that aggregate have finished transmitting. In this case, the aggregate is removed from the database (i.e. Aggregate Db), and the VLAN tag used to mark the aggregate is added back to the pool of available VLAN tags (lines 3 - 4 in Algorithm 2). The second type of OF rules that are removed are used to forward individual traffic flows. If a traffic flow stops transmitting, the corresponding OF rules become inactive and after five seconds they are removed. In this case, the Aggregation module removes the traffic flow from the Flows Db (line 6). The third type of OF rules that are removed also corresponds to individual traffic flows. However, unlike for the second type, these rules are removed because the traffic flows have been aggregated at some point during their transmission. When this aggregation occurs, new OF rules (i.e. aggregate rules) with higher priority are installed in the devices such that the traffic flow is forwarded as an aggregate instead of as an individual flows. Due to having higher priority, the aggregate rules forward the traffic flow while the old OF rules expire and are removed. When these OF rules are removed, no other action is needed since the corresponding traffic flow still exists, but is now forwarded using the OF rules for the new aggregate.

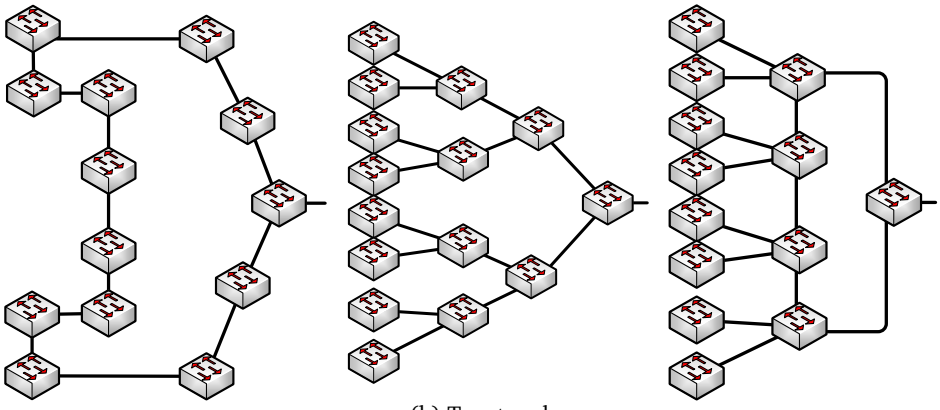
Algorithm 2 Aggregation algorithm: Flow removed event

```
1: rule ← removed OF rule
2: if rule == aggregate rule then
3:     remove the aggregate from database
4:     update VLAN pool
5: else if rule has not corresponding aggregate rule then
6:     remove flow from flows database
```

4.3 Evaluation and results

The purpose of the evaluation is to assess the performance of the proposed solution in reducing the number of OF entries in the forwarding devices. For this, the aggregation service is tested on three backhaul specific topologies (Figure 4.3). The first topology is a ring topology, the second is a tree, and the third is a combination of the first two, which is termed *ring of trees* here. It is expected that the choice of topology will impact the performance of the aggregation service.

These topologies were selected based on real backhaul deployments from an industry white paper [1]. All the backhaul topologies used here, transport traffic from the access networks to the core, hence they have one node which represents the gateway node towards the core network. This node is the farthest right in the topologies (Figure 4.3. Moreover, the gateway node has one host connected to it which is the destination for all the traffic flows in the network (i.e. traffic sink). The rest of the nodes in each topology have three hosts connected to them, representing the sources for the traffic flows. Three hosts were used in order to easily emulate three QoS classes by generating traffic with different DSCP marking from each of the hosts. Thus, the generic traffic pattern in each of the topologies is from left to right, having different DSCP markings, and all being destined towards the traffic sink host. The Linux *ping* command is used to generate the traffic flows. Even though the throughput of the ICMP generated traffic is not representative for a real scenario, it does not impact the assessment of the aggregation service, which takes into account only the number of traffic flows and not their size or duration.



(a) Ring topology (b) Tree topology (c) Ring of trees topology

Figure 4.3: Mobile backhaul topologies used for the evaluation

Mininet [6] is used to emulate the network topologies at the data plane, while the Floodlight based prototype described in section 4.2 represents the control plane. A testbed consisting of two Virtual Machines (VMs) is used here. One VM hosts the Mininet emulation and the other VM runs the SDNC. This split VM configuration allows for performance isolation between the emulated data plane network and the control plane logic.

The number of OF entries in the forwarding devices is used as a performance metric in this evaluation. Hence, the forwarding devices are polled continuously throughout the test by using OF *Table Statistics Requests* messages, in order to obtain

the number of OF entries in the devices. Based on these device specific measurements, the total number of distinct OF entries in the data plane devices is computed for each test. Two tests cases are executed for each topology: one without the aggregation service enabled and one with the aggregation service. Assuming that there are N_1 OF entries for the test case without the aggregation service, and N_2 OF entries for the case with the aggregation service enabled, the reduction in OF entries is calculated as showed in Equation 4.1. This value (R) is then transformed into percentage for the final results.

$$R = \frac{N_1 - N_2}{N_1} \tag{4.1}$$

Figure 4.3 shows the obtained results for all three topologies. Each test has been repeated with an increasing number of generated traffic flows per host in order assess the scalability of the aggregation service. Moreover, each test case (topology + number of traffic flows) was executed several times to guarantee the accuracy of the results. In the figure, the number of traffic flows is on the horizontal axis and the percentage of reduction is on the vertical axis.

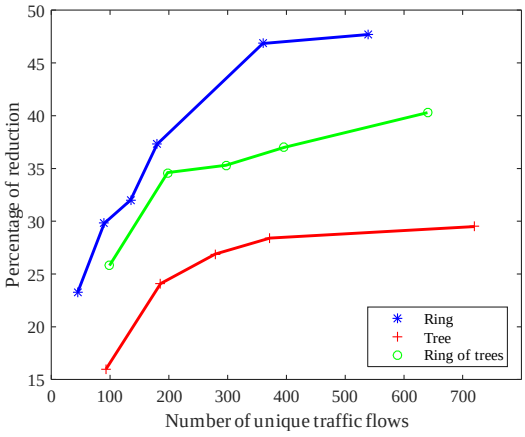


Figure 4.4: Percentage of reduction in number of OF entries by using the aggregation service

As it can be noticed, the aggregation service introduces a significant reduction in the number of OF entries in the forwarding devices, sometimes up to 50%. This means that only half the number of OF rules are needed to forward the same amount of traffic when the proposed aggregation solution is used, compared to the baseline case (i.e. no aggregation service).

Two patterns emerge from the obtained results. First, the aggregation service is more efficient with the increase in number of traffic flows, leading to a higher reduction when more traffic flows are generated in the network. However, it appears from the results that the efficiency of the aggregation service reaches a plateau for each of the topologies. This plateau represents the maximum reduction that can be obtained in a real scenario for the corresponding topology. This observation takes into account that in a real network the number of traffic flows will be much higher than in the emulated testbed used in here.

The second pattern in the results is that different topologies lead to different efficiencies for the aggregation service. Particularly, the tree topology has the lowest efficiency, while the ring topology has the highest efficiency. This behavior can be attributed to the inherent characteristics of these topologies. In a ring topology, there are only two paths that the traffic can take, in each direction along the ring. This leads to high path overlapping for the routes of the traffic flows, which is the main objective of the aggregation algorithm. In consequence, in the ring topology it is highly likely that more traffic flows are aggregated than for the tree topology, where the routes are spread to each of the leaf nodes. In this regard, the results for the ring of trees topology are in the middle, although closer to the ring topology.

Finally, the measurements showed very small standard deviation, in the range of 5 to 15 OF entries, which is less than 2% from the total number of OF entries. Thus, the obtained results are reliable and very accurate.

4.4 Discussion

A particular scenario where increased flow table size is needed in SDN, is when changing the network policies, combined with a *per flow consistency* requirement. This requirement states that a traffic flow must be processed in every device in the network by OF rules that correspond to a unique network policy. This requirement becomes difficult to fulfill when the network policies change, requiring that all the existing traffic flows must be processed by using the existing policies until they end, while the new traffic flows must be processed according to the new policy. In this case, there will be a certain period when the two network policies overlap thus requiring approximately twice the number of OF rules. If the devices cannot accommodate the increased number of OF rules, the traffic is processed incorrectly leading to unpredictable behavior in the network, and possibly service degradation (e.g. security violations, incorrect QoS provisioning, etc).

Traffic flow aggregation has other benefits apart from reducing the flow table size in OF device. For example, it is less complex to manage traffic in the network as aggregates rather than individual traffic flows. Specifically, when rerouting traffic from one link to another there are fewer OF rules to modify for an aggregate containing ten traffic flows than for the ten individual flows. Hence, traffic engineering is simplified.

Future research work could focus on enabling flow rerouting in the network to increase the path overlapping thus achieving better efficiency for the aggregation.

4.5 Chapter summary

As outlined in the beginning of the chapter, a challenge in the adoption of SDN for real network deployments is the reduced table size that the OF devices usually have. This is because the underlying technology, TCAM, is very expensive and power demanding, leading to expensive devices (purchase and operation). Moreover, the limited space in the flow tables can also affect the network services provisioned to the customers. As an example, service provisioning can be delayed until resources (i.e. flow table space) become available.

This chapter proposed a solution to address the challenge of limited flow table size in OF based SDN environments, which does not require any changes in the device architecture or the OF protocol. Specifically, the solution is based on dynamically aggregating traffic flows in the network, so that the same amount of traffic flows can be forwarded by using significantly fewer OF rules. By aggregating traffic flows, the SDNC may lose visibility and fine grained control over the traffic. Hence, QoS provisioning for the traffic flows may be hindered by the aggregation. To preserve correct QoS provisioning in the network, the proposed aggregation service takes into account the QoS class of the traffic flows by considering also the DSCP marking when aggregating the flows.

Section 4.3 provided conclusive performance results for the proposed solution, showing that in certain cases a reduction of almost 50% in the number of OF entries can be obtained by enabling the aggregation service. A key point emerging from the results is that the efficiency of such a solution highly depends on the topology considered. In this respect, a ring topology gives better results than a tree topology, with approximately 20% increase in efficiency.

CHAPTER 5

Control plane challenges for QoS provisioning in SDN

Continuing from the work presented in the previous chapters towards completely exploring the problem stated initially (section 1.4), this chapter investigates SDN control plane specific challenges that may hinder the provisioning of QoS in communication networks. The work presented in this chapter is based on Paper E and Paper F.

The key aspect in SDN is that the control logic, which is in charge of traffic processing decisions, is centralized in the SDN Controller (SDNC). Centralizing the control logic raises concerns about the performance of the controller when the underlying network grows very large. When the SDNC reaches a performance bottleneck, it cannot process control plane events fast enough, leading to longer network service provisioning times, reduced network throughput, etc. Hence, seen from a holistic perspective, the performance bottlenecks at the SDNC impact the QoS for the network services.

SDNCs comprise multiple functional entities for network control and management such as monitoring, routing, policy enforcement, logging, etc. All these functions are simply termed *control functions* here, since they belong to the controller, even though some of them may implement management features. Figure 5.1 illustrates the idea presented above, by collectively representing the control and managements functions within the SDNC. The *efficiency of a control function* denotes how well the function achieves its intended purpose. For example, a monitoring control function is efficient when it maintains an up to date view of the network resources. For a control function to be efficient, is must execute a set of operations, which consume from the computational resources available at the SDNC. Consumption of computational resources is termed *computational load* in this chapter.

Recalling from chapter 2, there are two types of behaviors for OF entries installation in SDN: *proactive* and *reactive*. When the SDNC has a *reactive* behavior, it must react to a multitude of traffic flows that arrive from the network, at the D-CPI. In this case, an important challenge is the scalability of the SDNC when the traffic flow arrival rate increases significantly[20]. With the increase in the number of traffic flows, the computational load at the SDNC increases as well. If the SDNC cannot scale when the computational load is high, a performance bottleneck arises, leading to poor scalability for QoS provisioning (e.g. increased flow setup delay).

Several existing works propose innovative solutions to reduce the computational load per SDNC instance, for example by deploying multiple controller instances. In this chapter, a new solution is proposed to address the same challenge of eliminating or reducing the impact of controller performance bottlenecks in SDN. The proposed solution aims to achieve a compromise between the efficiency of the control functions and the performance of the SDNC, when the computational load increases. It is important to emphasize that the challenge considered in here is particularly important to OF based, reactive SDN deployments. Alternatively, for proactive controller behavior, this challenge is minimized since the traffic flows in the network are not sent to the SDNC for further processing.

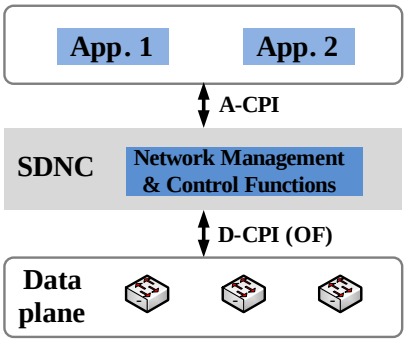


Figure 5.1: Generic SDN architecture

This work exploits the correlation that exists between the efficiency of the control functions that are hosted by an SDNC and the performance of the SDNC itself. This correlation is denoted by the computational load at the SDNC as explained in the following. Some control functions need accurate network information to correctly execute their logic. In a dynamic network, the state of the network and the traffic characteristics change frequently. Thus, some control functions must execute their associated operations very often in order to follow these frequent changes. Nevertheless, while executing these operations very often results in a good efficiency of the control function, they also drain the computational resources of the SDNC. This observation is especially valid for control functions that are computational intensive (e.g. path computation). When the computational resources are limited, the SDNC processes the control plane events much slower, hence reaching a performance bottleneck. In this case, the perceived QoS of the network services is degrading: the service provisioning is less responsive, the quality of the resources allocation degrades as well since various network statistics may be processed with delay. As a result, when limited computational resources are available, the SDNC must reach a trade-off between a good efficiency of the control functions and a good performance of the overall controller.

The proposed solution is to reduce the activity of the control functions in order to alleviate the SDNC performance bottlenecks. By reducing the activity of the control functions, fewer operations are executed and the computational load decreases. The specific challenge that is addressed here is to prove that the proposed technique can be applied with small or no impact on the efficiency of the control functions. An important requirement for this solution is that the functions are configurable, that is they expose their internal parameters for dynamic configuration at runtime. This allows for the *optimal configuration* to be applied on the control functions without interrupting the execution of the SDNC.An optimal configuration is defined here as the best trade-off, resulting in reduced computational load at the SDNC and a good efficiency of the control functions.

The rest of the chapter is structured as follows. Section 5.1 gives a introduction into various techniques proposed in the literature for increasing the performance of the SDNC when the network scales. Continuing, section 5.2 describes the prototype implemented here to demonstrate the validity of the proposed solution. Section 5.3 presents the evaluation of the prototype and discusses the obtained results. Finally, the last section summarizes the key findings in this chapter.

5.1 Background and related work

An important factor that impacts the scalability of the SDNC and has not been mentioned to this point in the chapter is the *protocol overhead*. This factor affects the load on the control channel, such that it may cause congestion at the D-CPI, severely affecting SDNC's performance. Moreover, OF messages can experience a high latency on the control channel further affecting the performance of the controller. In order to reduce the latency, the SDNC must be optimally placed in the network. The problem of optimally placing the controller in the network has been studied previously, resulting in several proposed solutions [9] [10].

Nevertheless, the most important cause for SDNC performance bottlenecks when the networks scales up is the limited CPU resources available at the controller [20] [5]. In an OF based SDN, the OF control messages, sent or received by the SDNC on the D-CPI, are associated with processing inside the controller. Specifically, the received messages trigger controller events that require processing (e.g. path computation, statistics update, etc.), and the sent messages are also associated with processing (e.g. statistics request). Hence, when the rate of OF messages is high, there are more controller events that must be processed inside the SDNC, consuming the available CPU resources.

There are two metrics that are commonly used in the literature to assess the SDNC's performance in an OF based scenario [18]. First, the *flow throughput* represents the number of traffic flows that the SDNC can process per time unit. This parameter is usually assessed by sending *Packet In* messages to the SDNC (at the D-CPI) at an

increasing rate. The second metric, termed *flow setup delay*, represents the time it takes to install OF entries, usually measured under low SDNC load. These two SDNC performance metrics are impacted by how fast the SDNC can process (i.e. generate responses to) incoming OF messages. The time for processing OF messages depends on other operations that are executed by the SDNC and also by the number of OF messages. This is because the available computations resources are shared between all these operations. Hence, the SDNC performance, as expressed through the above mentioned metrics, can be lower when the computational load is very high, affecting the traffic flows, impacting the network utilization and ultimately the customer or the user of the network services, leading to poor perceived QoS.

Several solutions have been proposed to address the challenge of controller performance in SDN. In general, the strategies for preserving the performance of the SDNC when the network scales up, can be split in two categories: *horizontal scaling* and *vertical scaling*. Horizontal scaling strategies imply that more controller instances are added to the control plane. On the other hand, vertical scaling strategies rely on increasing the computational resources of individual controllers. In the following, a major part of the solutions proposed in the literature are described.

5.1.1 Horizontal scaling strategies: flat architectures

Several controller instances that communicate with each other are used in this strategy, therefore enabling a unified control plane. [12] describes such a scaling strategy based on a distributed control plane, addressing several issues in SDN: scalability of the entire architecture, reliability and control plane performance. Their proposed controller, Onix, comprises several distributed controller instances, which maintain a consistent view of the network. This allows for the control plane related processing to be shared among the instances, thus eliminating eventual performance issues when the network grows large. However, the solution described in [12] deals only with static scenarios where the number of controllers is preconfigured thus being inflexible in coping with network changes. An immediate question that arises is, how to dynamically configure the number of controller instances (i.e. scale the control plane) when the network grows, or the number of traffic flows increases? A possible answer to this question is offered in [5], which proposes a solution to load balance the switch connections among the available controller instances. Nevertheless, their results focus only on the switch migration aspect without considering the impact of adding additional controller instances.

Another example of a distributed control plane in SDN is in [17], which presents the HyperFlow architecture. In HyperFlow, the network is divided into smaller domains. Further, each of these domains is managed by an SDNC, where the SDNCs coordinate among themselves, across the WAN, through a distributed file system. In this way, each SDNC can form a unified view of the underlying network comprising the smaller domains. In a similar fashion, [2] describes a distributed SDNC which runs multiple

controller instances that use an in-memory data bus to synchronize among themselves and maintain a unified view of the network.

The mechanisms presented above reduce the number of OF messages per controller to keep the computational load at the SDNC within acceptable boundaries. This is achieved by dividing the total amount of OF messages to be processed, among multiple SDNC instances.

5.1.2 Horizontal scaling strategies: hierarchical architectures

There are other horizontal scaling strategies that differentiate from the ones presented in section 5.1.1 by the various roles that the SDNC instances have, giving rise to a hierarchical structure at the control plane. [8] is an example of hierarchical controller architecture, where a root SDNC manages several local controllers. Moreover, the SDN services are of two types, services that need a view of the entire network (e.g. traffic engineering), and services that need only local knowledge (e.g. based on local traffic events). The former type of services are executed in the root SDNC, while the latter type of services are executed in the local controllers.

A new D-CPI protocol, named Devoflow, is proposed in [3]. Devoflow aims to reduce the computational load at the SDNC by keeping the traffic flows in the data plane as much as possible. The SDNC has control over the most important traffic flows such as long lived and large flows. A complete view of the network is realized by aggregating statistics from the forwarding devices. Given this, Devoflow creates a hierarchy for the traffic types, thus reducing the SDNC computation load by sending only the most relevant flows to the control plane.

New types of forwarding devices (termed *authority switches*) are proposed in [21], along with the OF forwarding devices. Authority switches keep the rules for how the traffic should be handled. Whenever an OF device does not know how to process a traffic flow, it redirects the flow to an authority switch. In this architecture, the SDNC distributes the rules for traffic processing to the authority switches. This design keeps the traffic flows in the OF devices and authority switches, eliminating the load on the SDNC thus achieving better scalability.

5.1.3 Vertical scaling strategies

Most of the enterprise applications today are hosted in cloud infrastructures, which are often purchased from service providers (e.g. Amazon [1]) or even privately owned by the enterprise. In such cases, the applications are executed on Virtual Machines (VMs) that are deployed in powerful servers. Executing applications in a virtualized environment (i.e. VM) brings increased flexibility for allocation of computational resources to applications. As an example, it is possible to dynamically allocate more virtual CPUs (vCPUs) or more memory to a VM. This procedure of increasing/decreasing the computational resources of a server or VM is named *vertical scaling* [13].

An SDNC can be executed as a regular software application in a VM hosted by a powerful server. By doing this, it is possible to leverage the benefits of server virtualization, as emphasized above. Specifically, computational resources (vCPU and memory) can be dynamically allocated to the SDNC, to cope with rises in computational load. Adding more resources to the hosting machine, allows the SDNC to process the control events faster, leading to lower flow setup delay and increased flow throughput.

There has been significant research devoted to intelligent strategies for vertical scaling [6] [4]. The fundamental idea for these strategies is to automatically adjust the VM resources (i.e. increase/decrease) according to the computational load. Test results in the literature [6] [4] confirm that vertical scaling is faster than horizontal scaling in coping with performance bottlenecks. Specifically, it is faster to increase the amount of resources allocated to a VM than starting a new VM and running an additional application in it. Moreover, there are software applications that are suitable only for vertical scaling, making other strategies unsuitable. Nevertheless, a fundamental limitation of vertical scaling is that the computational resources are always limited by the existing resources of the physical server. Hence, a VM cannot get more resources than what the physical server hosting the VM can provide. Another limitation is sharing these resources with other VMs that are hosted by the same physical server, potentially causing performance interference between the VMs. Finally, there is no guarantee that the server virtualization platform supports resources allocation at runtime, without having to interrupt the SDNC which would bring additional challenges for the network.

5.1.4 Controller configuration optimization strategy

Some of the solutions presented above are very flexible in coping with controller performance bottlenecks by increasing the computational resources of the hosting machine. However, no results are provided with respect to the dynamic scaling process in an SDN architecture. To be precise, no assessment is provided about the time required to add a new SDNC instance to offload the computational load from existing instances. Instead of focusing on the underlying software and hardware platform that hosts the SDNC, the solution proposed here focuses on the architecture and the behavior of the SDNC itself. Precisely, in this work, the SDNC performance bottlenecks are mitigated by optimally configuring the operational parameters of the control functions inside the SDNC. This strategy imposes additional requirements upon the SDNC architecture but also makes the SDNC scaling strategy independent of the underlying platform. Another important benefit of the proposed solution is that it can co-exist with the vertical and horizontal scaling mechanisms.

Since the approach presented here does not rely on complex operations for scaling the SDNC, it is more suitable for mitigating performance bottlenecks that are temporary. For these temporary (short lived) bottlenecks, which may be due to short spikes in the computational load (e.g. increased number of flows arriving in the network), the more complex operations are not very suitable. Strategies like creating a new

SDNC instance or allocating more resources to the existing SDNC are a better fit for alleviating performance issues that are long term in their nature.

While it decreases the computational load at the SDNC, reducing the activity of the control functions also has a negative effect of on the efficiency of functions. This represents the key aspect that is investigated in this chapter. Following, there are examples of this negative effect for a number of control functions, to emphasize the spectrum of choices and optimizations that the proposed solution has:

- *Network monitoring* is realized by continuously fetching statistics from the network. For OF based SDN, a polling mechanism is used to monitor the network, by sending *Statistics Request* OF messages to the forwarding devices. Decreasing the sending rate of the request messages results in less accurate network statistics, which impacts the efficiency of the monitoring function itself, and also the efficiency of other control functions that rely on the monitoring information.

- Routing, especially load-aware routing or path computation, is based on regular computations of routes that reflect the traffic load in the network. Decreasing the rate of route computations leads to decreased efficiency of the routing function and further to reduced network utilization.

- Another control function that requires regular operations is *policy based traffic management and control*. For this function, policies are continuously verified and validated against the network conditions and among themselves for conflicts. Reducing the rate of these operations may result in temporary inconsistencies with respects to the traffic or network status (bandwidth is above the allowed limit, inconsistent routes in the network, etc.).

- A common control function in real life SDN deployments is *logging*. This function implies gathering performance metrics, statistics about events from the entire network and keeping this information for future investigation. When the computational load is very high at the SDNC, this function could be configured to log only critical events for a short period of time in order to overcome the bottleneck. While possible, this can lead to an unacceptable loss of information about the status of the network.

- SDNCs can host various control functions that perform packet analysis, by processing a large amount data packets and network events, and correlate the information to detect critical behavior (e.g. network attacks). If the rate for packet and event processing is decreased, such a control function may omit to detect critical events and take appropriate action.

In this chapter, the feasibility of the proposed strategy for SDNC performance bottleneck mitigation is assessed. To this end, a series of test have been performed

with the prototype implemented and described in the next section. Two control functions have been considered for the assessment: network monitoring and load-aware routing, which are common in SDN implementations and they also consume a significant amount of computational resources.

5.2 Prototype implementation

A prototype has been implemented in order to assess the idea proposed in this work. The prototype architecture depicted in Figure 5.2, and it is based on Floodlight controller [7]. As in the previous sections, some of the modules for this prototypes have been reused from Floodlight while others have been newly implemented. Moreover, only the modules that are relevant for the current chapter are depicted in Figure 5.2.

At the lowest level in the SDNC there is the *OF* module, which implements the OF based communication with the forwarding devices in the data plane. Further up, there is the *Forwarding* module which fetches the routes that are kept in the *Routing Information Base* (RIB) and configures them in the data plane by installing OF rules in the devices along each route. These two modules have been reused from Floodlight. The rest of the architecture is built around the two control functions that are considered in this work: *Monitoring* and *Load Aware Routing* (LAR). The modules implementing these two control functions allow for their internal mechanisms to be configurable through the REST API of the SDNC, as it will be explained in the following sections.

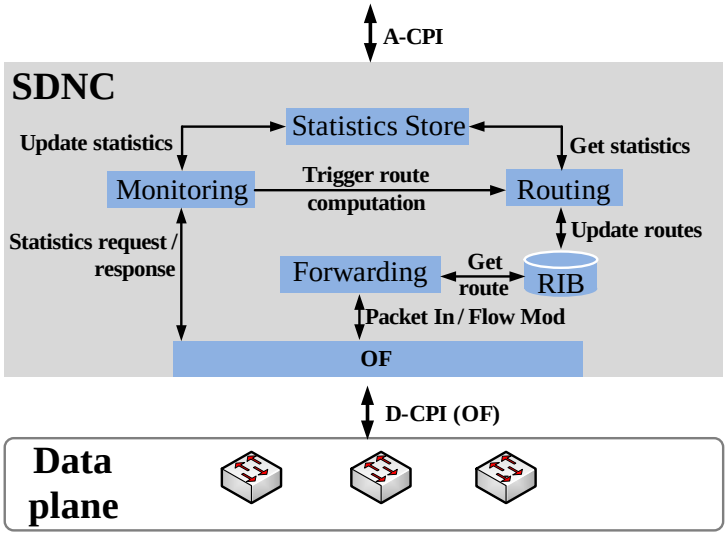


Figure 5.2: Prototype architecture

5.2.1 Monitoring Module

The purpose of the Monitoring module is to gather statistics about the network and keep them in the *Statistics Store*. Inside the Statistics Store, the monitoring information is kept as a collection of data objects, which are shared (i.e. accessed) by the Monitoring and Routing modules. There are two statistics that are collected for the current implementation: *link bandwidth utilization* and *link delay*. Statistics are collected periodically, every *t* seconds (default 1 second), where the *t* parameter can be configured through the REST API. Configuring the rate of statistics requests through the *t* parameter, impacts the accuracy of the network monitoring information in the Statistics Store.

Bandwidth utilization on the links is measured through a polling mechanisms by sending *Port Statistics Request* OF messages to every forwarding device. Apart from bandwidth monitoring, these messages are also used to compute the delay that a packet usually has on the control channel (i.e. at the D-CPI interface). This delay is used later in the network link delay measurement procedure, which is implemented as a custom mechanism, based on [16]. Specifically, for the link delay measurement, the SDNC sends packets with timestamps on all the links in the network. Such a packet travels one link and, at the next hop (forwarding device), it is sent back to the SDNC. Inside the SDNC, the Monitoring module records the arrival time of the packet and computes the end-to-end delay by subtracting the initial timestamp. This end-to-end delay comprises the time spent by the packet on three links: SDNC-to-device, device-to-device, and device-to-SDNC. Thus, the end-to-end delay includes also the time spent by the packet on the control channel. To eliminate the impact of the control channel delay, which is already estimated from OF Port Statistics Requests, this value is subtracted from the computed end-to-end delay, leading to better approximation the network link delay. This method for delay measurement is influenced by the computational load at the forwarding devices, since the measurements comprises also the processing time inside the devices. Nevertheless, the results in [16] show that by using the technique described above the delay can be measured with an accuracy comparable to the ICMP based measurements.

The Monitoring module also detects significant changes in the network traffic load. When a significant change occurs, the Monitoring module triggers the route computation algorithm in the Routing module (Figure 5.2). This is to ensure that the routes are up to date with respect to the network load, for a better traffic optimization. A *triggering policy* denotes when a change in the network traffic load is considered significant. Hence, depending on the triggering policy, the SDNC has a varying level of sensitivity to network changes, further influencing the rate of route computations. Two triggering policies are implemented in this prototype. The first policy is based on the measured delay and the second is based on the measured link bandwidth utilization. Figure 5.3 illustrates how the triggering policies are implemented, by using as an example the triggering policy based on the measured bandwidth utiliza-

tion. Specifically, the difference between two consecutive measurements for the link bandwidth utilization is computed. If this difference is higher than a certain threshold (termed *bandwidth threshold*), then the trigger is activated. There is only one threshold per metric (link bandwidth utilization and link delay)for all the links in the network, and both are configurable through the REST API exposed at the A-CPI allowing for a dynamic control over the sensitivity of the SDNC.

The amplitude of the measured instantaneous statistics can vary greatly, having sudden spikes which may result in triggering the route computation very often and unnecessarily. To alleviate the effect of these measurement spikes, the Java API for reading the statistics from the Statistics Store automatically averages the statistics over *N* measurement samples, where N is configurable through the REST API.

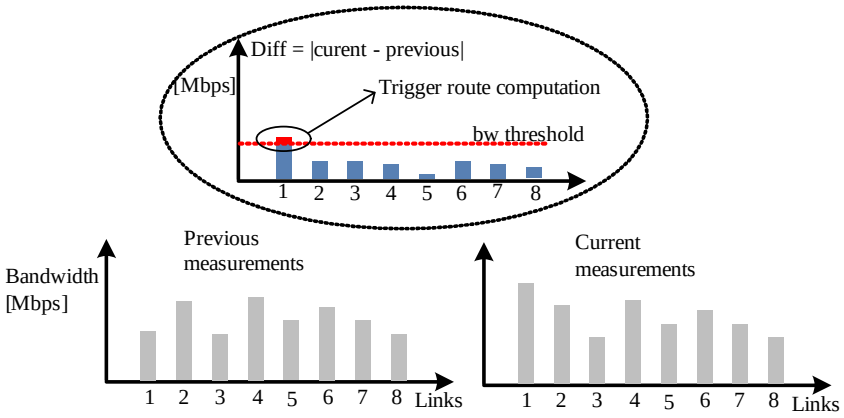


Figure 5.3: Graphical representation of the bandwidth threshold parameter

With respect to network monitoring, the Monitoring module and the Statistics Store enable a decoupling between the network and the client applications that need monitoring information for their logic. One one hand, there is a loop for statistics gathering between the Monitoring module and the data plane. On the other hand, the Statistics Store offers processed monitoring information to upper layer SDN applications. Additionally, the client applications can use the REST API to control the accuracy and averaging of the monitoring information kept in the Statistics Store.

5.2.2 Routing Module

The Routing module implements a Load Aware Routing (LAR) algorithm. The logic of this algorithm is based on the network monitoring information that is kept in the Statistics Store. After the computation is done, the Routing module outputs a set of routes which are then kept in the RIB (Figure 5.2). Each route computation is triggered by the Monitoring module according to the triggering policies.

In the current implementation, LAR is based on Dijkstra's Shortest Path First (SPF) algorithm. However, instead of using the basic hop metric, two QoS performance metrics are used in here: the *bottleneck bandwidth* and the *link delay* (i.e. queueing + transmission + propagation delay). The *bottleneck bandwidth* for a route is computed as the minimum residual bandwidth for all the links that belong to that route, where the residual bandwidth is the capacity of a link minus the bandwidth utilization for that link. Based on these metrics, the LAR algorithm computes the *Shortest Widest Path* (SWP). In particular, for a given source and destination it first computes the path with the largest bottleneck bandwidth (i.e. widest). If multiple paths are equally good, then it selects the one with the smallest end-to-end delay (i.e. shortest) [19].

Executing the LAR algorithm very often consumes significant amount of computational resources especially for large networks (e.g. hundreds of nodes). Hence, the frequency of route computations should be bounded in order not to exhaust the computational resources available to the SDNC. To this end, a *clamp down timer* (CDT) is implemented to govern the route computation process. CDT aims to limit the rate of route computations, concept which is also illustrated in Figure 5.4. Particularly, CDT denotes the minimum allowed time between two consecutive route computations. When CDT expires, new routes are computed only if there are any active triggers, or if a change in the topology occurred. Finally, the CDT parameter can also be configured through the REST API.

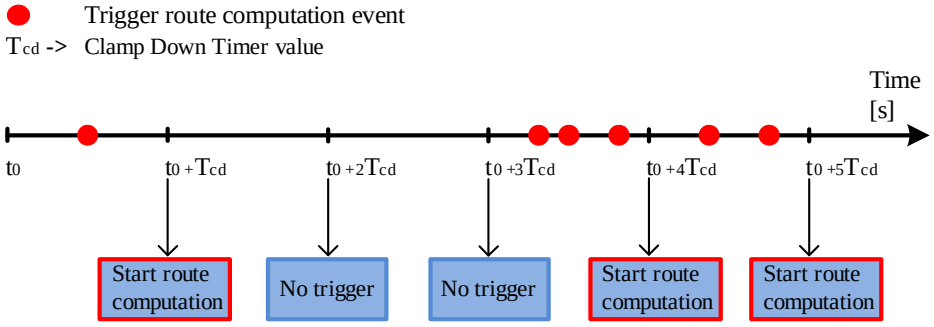


Figure 5.4: Graphical representation of the clamp down timer parameter

5.2.3 Testbed setup

Two tests are performed in order to validate the idea proposed in this chapter. To be precise, the aim of the tests is to prove that the computation load at the SDNC can be reduced through careful configuration of its functional entities, without a significant impact on the efficiency of the control functions. Each test assesses the impact of one configuration parameter on the two control functions implemented in the prototype,

LAR and real time network monitoring. The configuration parameters covered in the tests are the *bandwidth threshold* (Figure 5.3) and the *CDT* (Figure 5.4).

Both control functions considered in here impact the computation load at the SDNC since they require executing several operations with a high rate. Precisely, LAR has a cubic average time complexity in the number of nodes ($O(n^3)$). Moreover, network monitoring is computationally expensive mainly due to processing a large number of statistics and having concurrent access on the Statistics Store (multiple threads reading and writing statistics). In order to be efficient, LAR requires accurate monitoring information (i.e. high statistics requests rate) and frequent route computations. The methodology used for the tests is to decrease the SDNC's computational load by adjusting one of the two configuration parameters (i.e. bandwidth threshold and CDT). Next, the impact of these adjustments on the efficiency of the control functions is observed. To assess the efficiency of the control functions, the average network throughput is used as a metric. This metric reflects the efficiency of the LAR algorithm (routing control function), given that the purpose of this control function is to increase network utilization. Moreover, the proposed metric also reflects the efficiency of the monitoring control function, although indirectly, because the accuracy of the monitoring information affects the quality of the routes.

The work done in this chapter is highly applicable in network environments that have a large number of traffic flows such that, due to the large scale, the SDNC's performance is greatly impacted. Although there are several types of networks that fit this generic description (e.g. data centers, WAN), a WAN topology has been chosen to validate the proposed idea. Two topology sizes have been used for the tests. The first one is a small scale topology (Figure 5.5), and the second one is a large scale topology (Figure 5.6). The small scale topology resembles the WAN deployed by Google between its data centers. Additionally, the large scale topology is created by interconnecting two small scale topologies with four additional links. This makes the large scale topology still similar to a typical WAN with respect to the node degree (i.e. degree of interconnection).

Because Mininet is used, the two topologies consist of OVS software switches and virtual links to interconnect the OVS instances. Each network link has a capacity of 20 Mbps, which is much less than a real WAN link (e.g. 10 Gbps), but it is chosen to suit the virtualized environment used for the tests. Higher capacity links would require that more traffic is generated to saturate the network, exhausting the resources of the machines that run the tests. As it can be noticed in Figure 5.5, there is one host attached to each network device (OVS instance), resulting in 12 hosts from where the traffic is generated. This observation is also valid for the large scale topology in Figure 5.6 even though the hosts (24 hosts) are not depicted to keep the figure simple. A capacity of 40 Mbps is allocated to the host-to-network links. This allows the hosts to generate enough traffic to saturate the outgoing network links. Iperf [11] is used to generate the TCP traffic flows from the hosts.

Figure 5.7 illustrates the complete testbed setup comprising three functional

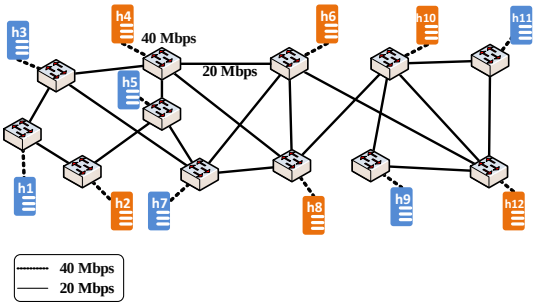


Figure 5.5: Small scale topology

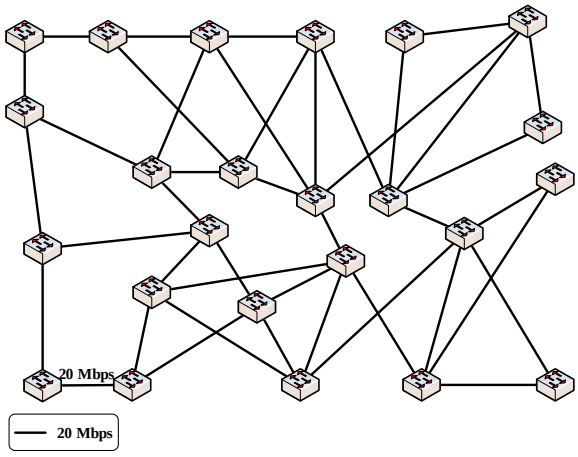


Figure 5.6: Large scale topology

entities deployed in two VMs. The first functional entity is the SDNC, the second is the Mininet test network, and the third is the test control script. During a test, the test control script uses the REST API to configure the operational parameters of the SDNC. Additionally, the test script also coordinates the test execution (i.e. traffic generation, etc.) by using the Secure Shell (SSH) protocol. Since Floodlight and Mininet consume the most resources and they also represent the test targets (control and data plane respectively), they are hosted in separate VMs to avoid performance interference among them. Both VMs are hosted in a physical server with 16 GB of RAM and one Intel Xeon E5-1620 CPU with 4 physical cores (8 vCPUs) running at 3.6 Ghz. The virtualization platform used to create and execute the VMs is Oracle VirtualBox [15], each VM being provided with 3 vCPUs and 4 GB of RAM.

Algorithm 3 captures the test execution logic contained by the test control script.

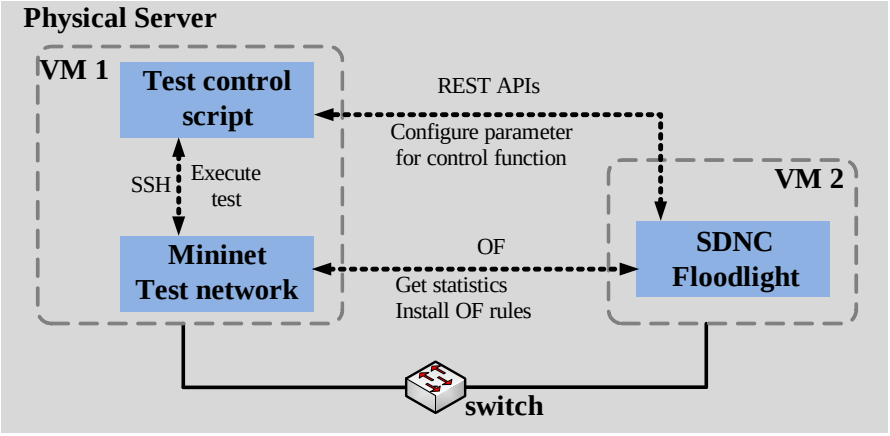


Figure 5.7: Testbed setup

Lines 1 to 3 contain the configuration variables for the test. In order to compute the standard deviation and understand the relevance of the results, each test is executed ten times, the *no_of_rounds* variable therefore being set to 10. In line 2, a list is created which comprises the configuration parameters values to be tested. These parameters are the bandwidth threshold and the CDT. Each parameter is assessed separately by varying its value while the other parameter is set to a default value. Line 3 contains the list of hosts, which serve as traffic sources and sinks. Depending on the test topology, small or large scale, N is equal to 12 or 24 respectively.

A configuration parameter value is first configured in the SDNC by using the REST API described in section 5.2 (line 5). Next, ten test rounds are executed for this configuration parameter value. Lines 7 to 11 describe the logic for the traffic generation. Each host in the network has a number associated with it (e.g. from 1 to 12). Based on the parity of the host number, two logical sets of hosts are created: odd hosts and even hosts. This is also illustrated in Figure 5.5 by the orange and blue hosts. While looping trough the set of hosts (lines 7- 8), traffic flows are initiated from a source to a destination only if they have the same parity, that is they belong to the same logical set (e.g. blue or orange). After a traffic flow is initiated, the test control script waits 1 second before generating the next traffic flow, allowing the SDNC to record the change in the traffic load produced in the network. In this way, the SDNC has sufficient time to follow the changes in the traffic load. Each traffic flow has a rate of 2 Mbps and a duration of 60 seconds. There are in total 60 traffic flows for the small scale topology and 120 flows for the large scale topology. The loop in lines 12-13 waits for the traffic flows to finish transmitting. In the final part of the algorithm (lines 14 to 15), the results for the current test round are computed and the testbed is

cleaned and prepared for the next test round.

Algorithm 3 Test control logic

```
1: no_of_rounds ← 10
2: config_parameters ← P_1...P_C
3: hosts ← h_1...h_N
4: for P_i in config_parameters do
5:     configure parameter P_i        // using the REST API
6:     for test_round=1 to no_of_rounds do
7:         for src_host in hosts do
8:             for dest_host in hosts do
9:                 if src_host != dest_host and have same parity then
10:                    start traffic flow from src_host to dest_host        // using SSH
11:                    wait 1 second
12:        while traffic flows are not done do
13:            wait 1 second
14:        compute results for this test_round
15:        clean testbed and prepare for next test
```

5.3 Evaluation and results

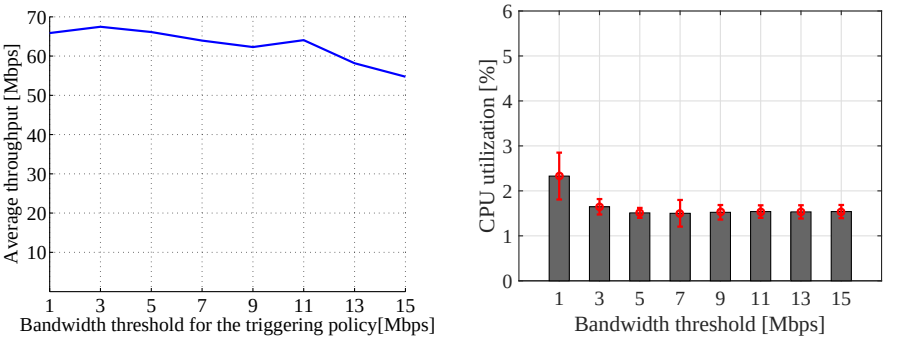
The aim of the first test is to assess the impact of the bandwidth threshold triggering policy, while for the second test the impact of the CDT operational parameter is evaluated.

5.3.1 Test 1

Theoretically, the triggering sensitivity affects the computational load at the SDNC as well as the efficiency of the two control functions, as reflected by the metric chosen (i.e. average network throughput). In this test, the bandwidth utilization triggering policy is evaluated in order to validate the idea proposed herein. Hence, the bandwidth threshold takes values from 1 Mbps to 15 Mbps, resulting in 15 triggering policies. These policies range from policies that are very sensitive to small changes in the network traffic load, to policies that are sensitive only to very big changes in the traffic load.

Figure 5.8 captures the results for the test on the small scale topology. On the left side (Figure 5.8a), the average network throughput can be observed, and on the right side (Figure 5.8b), the measured CPU utilization is plotted against the various bandwidth thresholds. As it can be noticed in the results, by increasing the bandwidth threshold the average network throughput gradually decreases. This

is explained by the fact that a higher bandwidth threshold results in less accurate routes in the RIB thus negatively affecting the network utilization. At the same time, while the bandwidth threshold increases the computation load, reflected in the CPU utilization, decreases (Figure 5.8b). The important observation is that the patterns for the two traces, average network throughput and CPU utilization, can be exploited to reach a compromise for configuring the bandwidth threshold when the SDNC is computationally overloaded. Specifically, the traces suggest that it is better to select a bandwidth threshold of 3 or 5 Mbps in order to reduce the CPU utilization (i.e. computational load), but still have a good efficiency of the control functions.



(a) Average network throughput

(b) CPU utilization with standard deviation

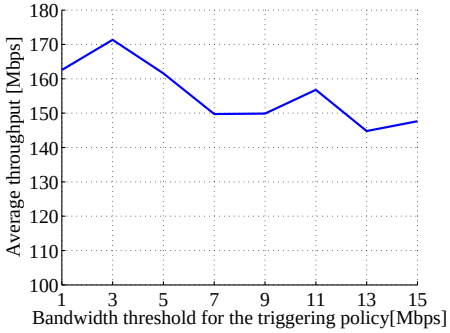
Figure 5.8: Results for bandwidth threshold for the small scale topology

The test results for the large scale topology are shown in Figure 5.9. In these results, the traces show the same patterns as emphasized above, but at a different scale. Compared to the previous results (small scale topology), the network traffic load is higher thus the measured average throughput is also higher (Figure 5.9a). Moreover, the CPU utilization is in average higher with 1-2% for the large scale topology (Figure 5.9b) than for the small scale topology (Figure 5.8b).

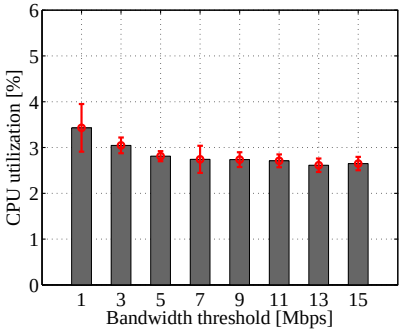
Even though the results follow the same pattern as in Figure 5.8, it can be noticed that for the large scale topology the decrease in CPU utilization is less prominent than for the small scale topology. This is due to the nature of contributions to the computational load at the SDNC. There are two main inputs given to the SDNC that contribute to the CPU utilization (increase the computational load). First, there is the size of the network topology with respect to the number of devices and links, which impacts the running time of the routing algorithm and the monitoring control function. For larger networks (i.e. with more devices and links), the route computation takes more time, following the cubic function ($N * O(N^2)$). Additionally, more *Statistics Request* OF messages must be processed by the SDNC. Second, there is the number of traffic flows arriving at the SDNC. The *Packet In* OF messages associated with

these traffic flows must be processed by several modules, consuming computational resources. By increasing the bandwidth threshold, only the rate of route computations is reduced. The amount of traffic flows (i.e. *Packet Ins*) processed and the rate of *Statistics Request* OF messages remain unchanged. Hence, in both figures (5.8b and 5.9b) the decrease in CPU utilization is only due to fewer route computations. In the second trace the average CPU utilization is higher with approx. 1% because of the contribution of monitoring a larger topology (24 instead of 12 devices), and processing twice as many traffic flows to process (120 instead of 60). With a higher average in CPU utilization for the second test, the decrease in CPU utilization, which is approximately equal for both cases (small and large topologies), is less prominent in the second case than for the the first one.

The observation made above implies that the CPU utilization can still be decreased if the rate of sending *Statistics Requests* to the forwarding devices is decreased. This rate can be decreased without further affecting the average network throughput since fine granular monitoring information (every second) is not necessary when the route computations occurs rarely (e.g. every 10 seconds).



(a) Average network throughput



(b) CPU utilization with standard deviation

Figure 5.9: Results for bandwidth threshold for the large scale topology

Overall, the CPU utilization for both set of results (small and large scale topologies) is low (smaller than 5%), although the control functions considered here are computational intensive. As mentioned, this is due to the small input given to the SDNC. To give a different perspective upon this behavior, the average rate of route computations for the test performed on the large scale topology is plotted in Figure 5.10. As it can be noticed in the figure, while the rate of route computations has decreased approximately ten times, the CPU utilization has decreased only 1% (Figure 5.9b). This confirms that the input given to the routing algorithm is not high enough to consume significant computational resources. It is expected that for much larger topologies (hundreds or thousands of devices) and a higher rate of arriving traffic

flows, the CPU utilization will have a much higher amplitude.

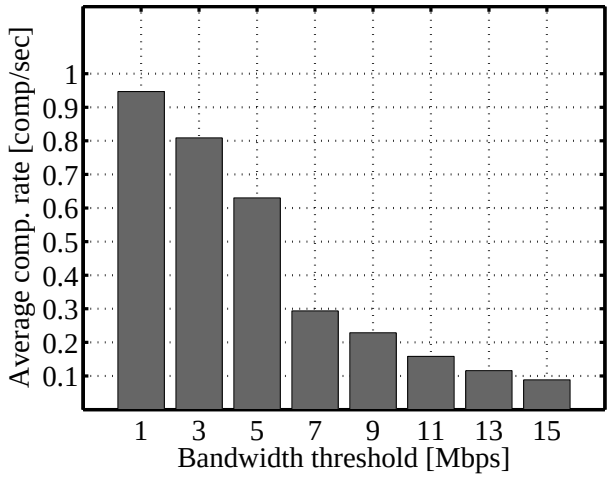


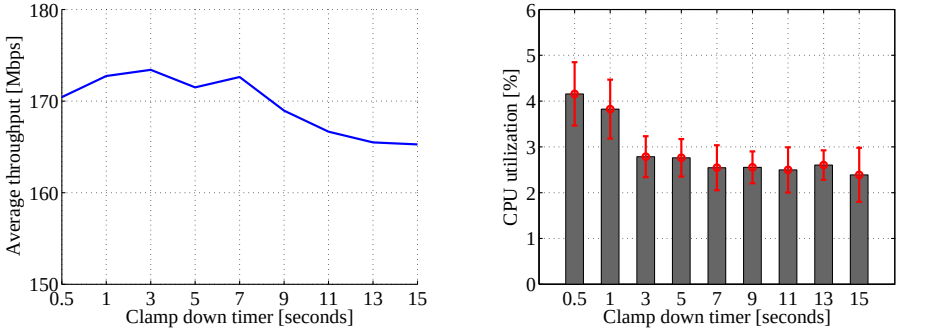
Figure 5.10: Average rate of route computations for the bandwidth threshold test on large scale topology

5.3.2 Test 2

In this test, the CDT operational parameter is evaluated. Specifically, the CDT parameter is varied in order to assess te possibility to achieve the best compromise between the efficiency of the control functions and the computational load at the SDNC. To this end, CDT takes nine values, from 0.5 to 15, as indicated on the horizontal axes in Figure 5.11. For all the tests performed in this section, the bandwidth threshold has been configured with the default value of 2 Mbps. This default value was chosen so that the route computation is triggered very often (high sensitivity to traffic load changes). Only the large scale topology is tested here (Figure 5.6).

Figure 5.11a shows the average network throughput plotted against the nine CDT values. The measured throughput increases slightly for the first three CDT values, then decreases gradually. This gradual decrease is expected and motivated by the fact that higher CDTs leads to more traffic flows being forwarded on the same route. Should this be the case, if the CDT value is very high (e.g. one month), a unique set of routes will be used for all the traffic flows, hence not using the multiple possible routes in the network. In this extreme case, the throughput will be much lower than the throughput measured for the results presented here.

The CPU utilization decreases in a similar fashion as for the bandwidth threshold tests. However, the important thing is the correlation between the CPU utilization trace



(a) Average network throughput (b) CPU utilization with standard deviation

Figure 5.11: Results for bandwidth threshold for the large scale topology

and the average network throughput trace. This correlation must guide the SDNC in choosing the optimal configuration for the operational parameters. In this particular case, selecting a CDT of 3 seconds leads to a sudden decrease in CPU utilization (i.e. computational load) and, at the same time, a higher average throughput (better efficiency of the control functions). This observation validates the feasibility of the idea proposed in this chapter that there is an optimal configuration for the operational parameters of an SDNC, through which the temporary performance bottlenecks can be mitigated.

5.3.3 Discussion

Results indicate that for the network topology and traffic pattern investigated here, adjusting the CDT leads to better results than adjusting the bandwidth threshold. This is because a similar decrease in CPU utilization can be obtained by adjusting any of the parameters, but higher network throughput is obtained through the CDT adjustment.

It has been mentioned that both the measured average CPU utilization and the decrease in CPU utilization are low (1-2%). Two of the possible reasons for this behavior are:

- The SDNC is deployed in a powerful VM (section 5.2.3)
- The network size (number of devices) and the number of traffic flows were too small to overload the SDNC. Floodlight controller can handle flow rates much higher than the tests performed in here, using an identical server configuration [22]. To be precise, [22] reports that Floodlight can process up to 500000 traffic flows per second. However, in [22] the data plane has been replaced with a performance tool for benchmarking OF controllers [14]. On the other hand,

> in this chapter, a network emulator was used for the tests (Mininet), limiting the possibility of generating traffic flows with a very high rate. Nevertheless, emulating the real network allowed for generating real traffic and measuring the average network throughput in order to assess the efficiency of the control functions, which is critical to validate the idea proposed in here. This would not have been possible by using the performance tool for benchmarking OF controllers [14].

To strengthen the results, a real network testbed must be used, instead of an emulated data plane. Moreover, flows must be generated with a much higher rate. In this way the tests are scaled out, leading to more noticeable patterns in the results.

Another possible way to better emphasize the idea proposed in this chapter is to scale down the testbed. By doing this, even a smaller computational load, resulting from the topologies presented previously, can have a significant impact on the recorded CPU utilization. In consequence, the patterns in the results can be better emphasized. Nevertheless, if the resources allocated to the VM that hosts the testbed are reduced, than the VM itself can be easily affected by other interferences, resulting in less predictable patterns in the result.

To automate the entire configuration, the SDNC must be self-configurable. Hence, a specific module must serve as the core of the self-configuration process. This module should monitor the status of the network to infer the efficiency of the control functions. In the test herein, the efficiency of the control functions is assessed by computing the average network throughput since this performance metric directly reflects how well the routing algorithm (i.e. LAR) is performing. and the quality of the monitoring information. Another metric that could be used is the packet loss in the network, or the average delay for the traffic flows. However, for other control functions, new metrics must be defined to assess their efficiency. Ultimately, since the aim of the SDNC is to provision network services with good QoS to various customers, metrics can be designed to take the customer feedback into account, such as customer or application Key Performance Indicators (KPIs). In this case, the configuration module in the SDNC must mitigate the performance bottlenecks by reducing the activity of the control functions, but still maintain good KPIs as reported by the customer or the application. Essentially, the goal is to mitigate the performance bottleneck while maintaining the level of QoS for the network services.

A significant drawback of having a centralized control plane logic, as SDN promotes, is that the SDNC is a single point of failure. This also has a negative effect on the QoS for network services, because service provisioning can be completely seized when the SDNC fails. However, it is common in real deployments that the SDNC is logically centralized, while physically it is implemented as multiple controller instances. For this reason, the challenge of having a single point of failure is SDN is not as important as the potential performance limitations at the control plane.

5.4 Chapter summary

This chapter analyzes the causes for SDNC performance bottlenecks. When the SDNC's performance degrades, the overall QoS for the network services also degrades. Specifically, services are slower to provision (long flow setup delay), network utilization is reduced due to SDNC not being able to follow traffic load variations, etc. It is important to mitigate the performance bottlenecks in order to prevent sudden degradations in QoS. There are already solutions such as vertical and horizontal scaling that solve the long term performance challenges at the SDNC. However, for short term and temporary performance bottlenecks, the existing solutions are not suitable due o their longer time scale (e.g. it may take minutes to provision a new controller). The solution proposed here mitigates the performance bottlenecks at the SDNC through careful configuration of the operational parameters for the control functions that are executed in the SDNC. The main objective when searching the optimal configuration for these operational parameters is to reduce the computational load, however still maintaining a good efficiency for the control functions. The results in this chapter show that the proposed idea can be applied to solve the performance problem when the network scales up.

While the proposed solution and the implemented prototype are based on OF v1.0 protocol, the concept can be applied to any OF version. The main requirement is that the control functions are implemented in such a way that they are configurable, at runtime, through simple parameters.

CHAPTER 6

Policy based network management in SDN

The network management plane is the third major area which influences the level of QoS for network services, after the data and the control planes which were discussed in the previous chapters. Network management operations usually require human decisions regarding the network's behavior. While this is beneficial given that humans can be in control of how the network behaves, it also bears several limitations that become even more significant as networks grow in complexity. First, it is well known that network operators encounter large expenses due to human error and misconfiguration [11] (pp.5). Second, human intervention usually requires more time to complete when compared to an autonomous system. This makes networks less agile in coping with changing needs. Moreover, service provisioning can be delayed if it requires humans to take decisions and configure network devices. Overall, the negative aspects of poor management mechanisms reflect into the level of QoS in communication networks.

A solution to alleviate the problem outlined above is to introduce more automation in the network management operations. Moreover, human intervention should be required only for setting high level network objectives. These high level network objectives can be specified in the form of *network policies*. Through automation, these network policies are processed and enforced in the underlying infrastructure, which continually adapts to changing traffic conditions and business needs.

This chapter is organized in two main sections. The first section demonstrates how policy based network management can be applied to solve a real world problem and achieve better resources utilization with less manual configuration. For this, a mobile network scenario is used and an SDN based framework for policy enforcement is employed in this scenario to solve challenges that network operators face nowadays. The second section concentrates on the characteristics policy frameworks: presents state of the art models for policy based network management, introduces the architecture of a novel policy framework, and illustrates its performance through a series of tests. Moreover, this chapter provides key insights about policy based network management, outlining what are the biggest challenges and potential solutions for developing policy frameworks that can be applied in real networks. Finally, the chapter emphasizes how policy based network management represents a key aspect for provisioning of QoS in

future communication networks.

The work presented in this chapter is based on two papers. Section 6.1 is based on Paper G and section 6.2 is based on Paper H.

6.1 Policy based traffic management in C-RAN mobile networks

A challenge that network operators face nowadays is managing the complexity of their mobile networks while, at the same time, leveraging their infrastructure to maximize the profits. With the adoption of over-the-top (OTT) services, Mobile Network Operator (MNOs) are bound to become just "bit pipe" providers by offering network connectivity with flat rates for sale. At the same time the traffic in their network grows, requiring more and more capacity and complex management operations (e.g. traffic engineering, security, packet inspection, etc.). To overcome this situation, network technologies are developed to increase the utilization of the network infrastructure and reduce the expenses for operating it. Such a technology is Cloud-Radio Access Networks (C-RAN).

C-RAN leverages the idea that a better resource utilization can be obtained if the physical resources are shared among the entities requiring those resources. This is the same concept that has been successfully applied in cloud computing, when multiple Virtual Machines share the resources of a physical server. To enable this idea in C-RAN, the functionality of a base station (BS) is split in two components: Remote Radio Head (RRH) and Baseband Unit (BBU), as illustrated in Figure 6.1. The RRHs are located next to the antenna, on site, and perform digital/analogue conversion and power amplification. Conversely, the BBU is centralized in a BBU pool, and comprises most of the BS functionality, particularly the digital processing of the signal [6]. With regard to the comparison with cloud computing infrastructures, the BBU pool resembles a typical data center, enabling sharing of physical resources among multiple entities. Due to this decoupling introduced in C-RAN, the RRHs are generic "technology agnostic" components, that are cheaper and easier to deploy. Additionally, the processing intelligence is centralized in the BBU pool, and shared between the RRHs depending on the traffic load variations. This architecture leads to savings in Capital and Operational Expenditures (CAPEX and OPEX) [6]. Further, it also creates new perspectives and service possibilities for equipment vendors, infrastructure owners and MNOs.

While C-RAN brings some immediate benefits as emphasized above, it also introduces new challenges. Particularly, C-RAN requires significantly more capacity in the fronthaul part of the network. The fronthaul spans from the RRHs to the BBU pool and transports In-phase/Quadrature (I/Q) data, requiring a ten fold capacity increase when compared to the cell capacity at the air interface [38]. Hence, even though CAPEX and OPEX can be reduced in C-RAN, the deployment of this technology can be prohibited by the high capacity requirements in the fronthaul. To facilitate

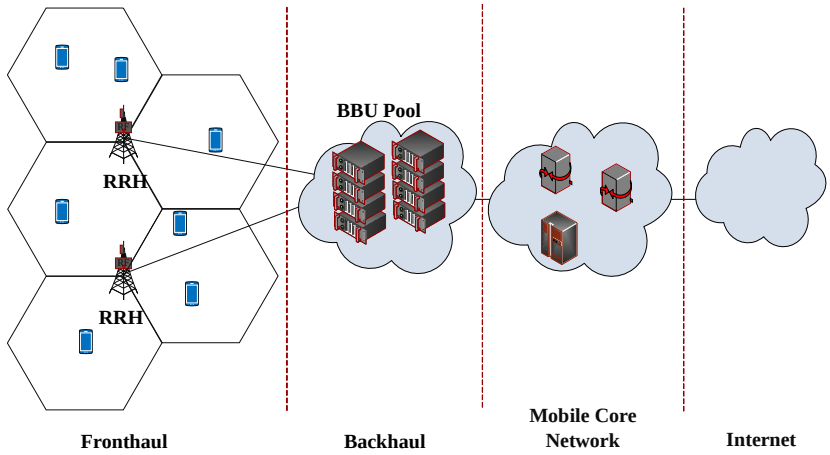


Figure 6.1: C-RAN architecture

the adoption of C-RAN, several MNOs can share a network infrastructure, which is potentially owned and operated by a third party infrastructure provider. This business model introduces a new entity often referred to as Mobile Virtual Network Operator (MVNO). A Service Level Agreement (SLA) is established between each MVNO and the infrastructure provider, which captures the characteristics and prices of the services offered by the infrastructure provider to the MVNOs. Through this business model, the infrastructure provider can use the network resources more efficiently, and the MVNOs can reduce their expenses since they do not own and operate a dedicated physical infrastructure. However, to enable this business model, infrastructure providers must be able to dynamically allocate resources to various MVNOs.

By centralizing the control plane logic in SDN, the controller maintains a global view of the underlying network, enabling it to allocate resources more efficiently than in traditional networks, which employ local decisions at the network device level [20]. This is a useful characteristic that infrastructure providers can leverage in order to maximize the utilization of the network resources towards enabling the business model described above. Moreover, given that SDN introduces programmability at the control layer, the decision logic of the network can be easily automated such that the resource sharing and management is done according to high level policies specified by human operators. Therefore, SDN enables autonomous management and dynamic sharing of the network resources. In this context, SDN is a feasible solution to support network infrastructure sharing in C-RAN environments. Nevertheless, there are some caveats that must be considered when applying SDN for network resources sharing in mobile networks, as it will be emphasized later in this section.

This section aims to demonstrate resource sharing in a C-RAN scenario by applying

the SDN paradigm. To this end, a resource sharing model is proposed here, and an automated policy based framework for network management is implemented to realize the proposed sharing model. Through this framework, an infrastructure provider can easily manage the network resource sharing by configuring high level policies that comply with the SLAs. The specific network resource considered herein is link capacity. Although there are other types of resources (e.g. radio resources) that must be shared in a real network deployment with multiple MVNOs, link capacity serves well enough to demonstrate the possibilities that SDN brings with respect to network management in general and policy based management in particular. To achieve this, the proposed framework is implemented and demonstrated on an emulated testbed. While this section emphasizes the overall benefits of applying SDN for policy based management, the next section in the chapter concentrates only on the policy based network management problem.

The structure of this section is as follows. First, an overview of the previous related research works is given. Next, subsection 6.1.2 describes the proposed resource sharing model, while subsection 6.1.3 presents the details of the implemented prototype. The evaluation results are reported and analyzed in subsection 6.1.4. Finally, subsection 6.1.5 concludes the work presented in this section of the chapter.

6.1.1 Background and related work

Given that OpenFlow (OF) has been initially designed to support Ethernet based TCP/IP communication networks, it had limited applicability to mobile networks, which employ mobile specific protocols (e.g. GTP). Even with the recent versions of the OF protocol, its applicability to the mobile domain is still limited. Nevertheless the concepts promoted by SDN, such as centralized control plane logic, received interest from the research community. In this regard, there are several works that attempt to apply the SDN paradigm to mobile networks, although with partial support due to the limitations of OF [21] [37] [8] [5]. In general, these works propose a complete redesign of the various parts of the mobile network architecture. The core architectural shift introduced in the related works is to execute the mobile specific control functions (e.g. Mobile Management Entity (MME)) over the SDNC, creating a flat mobile architecture [8].

A new protocol, termed MobileFlow, is proposed in [37] to better support the SDN paradigm in mobile networks, arguing that the features provided by OF are not sufficient. Together with the new protocol, the authors in [37] also introduce new types of forwarding devices, termed MobileFlow forwarding elements, which have mobile specific processing capabilities such as support for GTP and GRE encapsulation formats.

In [21], the integration of SDN into the core of the mobile networks is facilitated through a novel SDNC platform termed SoftCell. Considering that a mobile network contains a multitude of specialized devices for packet processing, the SoftCell SDNC

focuses on steering the user traffic flows through these specialized boxes so that the traffic is correctly processed. The steering is performed according to high level administrator policies. To implement the steering policies, OF software switches are deployed at each eNodeB on the sites, which is a similar approach as for the work presented in this section.

With respect to the radio interface in the mobile network, SoftRAN [17] proposes the decoupling of the control and data planes inside the radio nodes. In SoftRAN, a set of base stations that are geographically related is considered as a single resource pool. From this pool, the SDNC allocates resources along three coordinates: space, time and frequency.

Combining the SDN and C-RAN paradigms has received increased attention in the research community, and several works investigate this possibility [40] [10] [1]. SDN is also regarded as a potential solution to provide the demanded flexibility in 5G networks. For example, in [7], the authors study the applicability of SDN in 5G networks with respect to elasticity and scalability, suggesting that SDN can ensure the required elasticity in the mobile backhaul. Moreover, the authors see the applicability of SDN in mobile networks as an expected enhancement to the LTE architecture. In this regard, they propose an architecture for SDN based 5G networks which contains OF devices to aggregate traffic from eNodeBs towards the core network.

In the current section, SDN is applied to enable network capacity sharing in C-RAN. It is argued that the control plane centralization introduced by SDN fits very well with the C-RAN architecture, which proposes the centralization of the processing capability. Moreover, SDN promotes network programmability, which, as also suggested by previous research works, leads to a very flexible architecture in comparison traditional networks that require human intervention and configuration. This flexibility is readily applicable in managing the traffic in the proposed C-RAN infrastructure sharing scenario.

However, an important challenge arises when employing SDN for network capacity sharing in C-RAN, due to the different mechanisms to transport the user data. In C-RAN, the baseband traffic that is transported between the RRHs and the BBU pool (i.e. fronthaul) has a constant bit rate, given that the entire cell bandwidth is sampled, regardless of the real utilization of the bandwidth in the cell at a particular moment. Due to this, the capacity of the physical link cannot be shared by using mechanisms specific to packet based networks, unless a packet based transport is used in the fronthaul [2]. Since OF based SDN is designed for packet based communication networks, it is therefore necessary to approach the link capacity sharing in a different way in order to be able to apply SDN for capacity sharing in C-RAN. For this to be possible, the capacity sharing must be performed at a higher layer (with respect to the OSI protocol stack), so that the constant bit rate I/Q stream already transports a traffic mix which contains traffic flows from different MVNOs. This traffic mix should follow the high level policies and SLAs (between the MVNOs and the infrastructure provider). Based on this observation, a new capacity sharing mechanism is proposed here, which

overcomes the challenge outlined above by using Ethernet software switches to shape the traffic at each RRH site. In the proposed mechanism, traffic shaping is realized through OF based traffic control and priority queuing mechanisms.

There is a multitude of techniques for QoS control that are employed in mobile backhaul networks, which can be leveraged and applied to SDN based architecture. Specific examples include shaping, weighted round-robin, weighted fair queuing scheduling, etc [25]. Since the focus of this section is link capacity sharing for Ethernet ports, techniques such as color-based policing are a possible addition in order to deal with excessive traffic demands [25]. However, the current service proposal is based on traffic capping, leaving the handling of excess traffic for future upgrades.

6.1.2 C-RAN sharing service modelling

In a similar way to the work in [14], the current section first outlines some considerations about the proposed SDN service and the network topology, then it describes the assumptions for modelling the real network by using the emulated testbed environment herein.

Service description

The service considered in this work is an SDN based capacity sharing service for the C-RAN architecture. Two MVNOs are considered here in order to illustrate the proposed service. Each MVNO has an SLA with the infrastructure owner, stating the performance parameters for the network service purchased by the MVNO from the infrastructure owner. The focus in this section is on the QoS network performance parameters that are captured in the SLA, in particular link bandwidth. Through the SLA, both parties agree that the service performance parameters should be kept within specific boundaries, given that certain network conditions apply (e.g. offered load is not too high). If the promised boundaries for a parameter are very strict (e.g. tight delay budget), then the advantage is twofold. First, it is beneficial for the MVNO since the purchased service has predictable QoS, leading to better services offered by the MVNO towards its customers (i.e. with predictable QoS). Second, the infrastructure owner can charge more for the service with such an agreement.

There can be multiple granularities for the performance parameters captured by an SLA. For example, limiting the site capacity for an MVNO is a basic, coarse granular, performance parameter. This can be enhanced with more complex policies applicable to particular traffic types or user groups (e.g. gold/silver users). Finer grained policies may be outside the scope of the SLA between the infrastructure owner and the MVNO, as they relate more to traffic management within a certain MVNO. Moreover, the traffic load on a site varies throughout time. For example, cell sites that are located in business areas are more loaded during the working hours. During these "busy hours", sharing of the infrastructure's capacity becomes critical, and it must depend on the SLAs between the infrastructure owner and the MVNOs. Hence, the service must

be high level and flexible enough to capture the SLAs between the MVNOs and the infrastructure provider.

It is necessary to make some assumptions when modelling the service and the test network in order to suit the particular aspects of C-RAN, as it will be described later. The service implemented and demonstrated on the testbed described next, focuses on rate limiting (i.e. capping) the user traffic for each site, and for each MVNO. Based on the SLAs defined between the MVNO and the infrastructure owner, the service automatically enforces different traffic rates for the MVNO and the cell site. In doing this, the service takes into account the site type (i.e. residential or office) and the traffic fluctuations throughout the day. This implies that by using this service, the infrastructure owner can reduce the manual management operations in its network, and also better exploit the physical resources.

Modelling assumptions

The upper part of Figure 6.2, depicts the real network topology that is used to describe the proposed capacity sharing service. This topology is based on C-RAN and comprises 7 cell sites, where each site has 1 RRH and 1 User Equipment (UE) per MVNO, denoted as A (with blue) and B (with red) for simplicity. Each RRH is connected to the BBU pool, which is further connected to the Evolved Packet Core (EPC) containing the three signaling entities, Serving-GW, Packet Data Network-GW, and Mobile Management Entity (MME). Lastly, the EPC is connected to the Internet, which is represented by a cloud in the figure.

In this architecture, the UEs are emulated by using Mininet hosts, which employ the TCP/IP protocol stack. Hence, this emulation abstraction does not consider the lower layer protocols in LTE, which are used in the UE on one side and at the eNobeB (RRH) or BBU at the other side. Since the service proposed in this section performs traffic capping at Layer 3 (i.e. IP layer), the abstraction used here is considered reasonable. Another important difference between the real mobile network and the Mininet emulation is that in Mininet the hosts are conected directly to an OF forwarding device (switch), while in a real mobile network scenario the UEs (emulated here by hosts) have their traffic processed by multiple mobile specific devices before it reaches an OF forwarding device.

By considering the assumptions described above, the complete emulation model for the real mobile network is presented in the lower part of Figure 6.2. The entire architecture is emulated by using Mininet. Further, the EPC functionality has been removed and is not emulated given that the EPC does not impact the proposed capacity sharing service, which is targeted towards the fronthaul part of network. Additionally, the Internet is emulated through a single host (H-0). In the Evolved Universal Terrestrial RAN (EUTRAN) part of the network, the UEs have a corresponding Mininet hosts which emulates their behavior with respect to traffic generation. As mentioned in the previous paragraph, the lower layers in the mobile protocol stack are

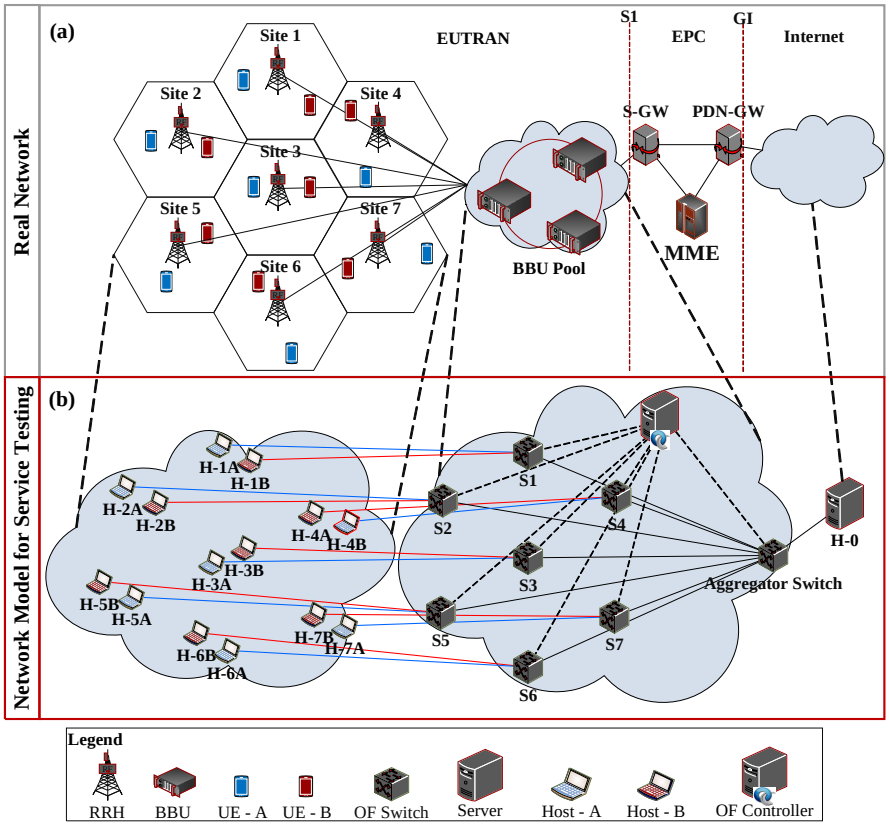


Figure 6.2: Mapping of the real mobile network (a) to the network model that is used for service testing in this section (b)

not included in the model. For this reason, the baseband traffic processing in the BBU pool is also not considered in the emulation. The BBU pool is therefore exclusively emulated with respect to the capacity sharing functionality which occurs at layer 3. To this end, the emulated BBU pool contains one virtual software OF switch (forwarding device) for each cell site, giving rise to an overlay network on top of the real BBU functionality. These OF devices that correspond to each RRH have a similar purpose as the OF devices co-located with the eNodeBs in [21] [7]. The aggregator switch is also an OF enabled forwarding device and its purpose it to aggregate the traffic towards the Internet, denoted by H-0. These OF devices in the BBU pool are used to rate limit the traffic (i.e. apply capping) for both Downlink and Uplink directions. Rate limiting occurs according to various policies that are configured at the SDNC,

which itself is also located in the BBU pool, close to the emulated overlay OF network. Each emulated host communicates with H-0, following the typical data sessions that exist in mobile network. Due to this assumption regarding the traffic pattern, it is easier to enforce rate limiting policies in the emulated network. Further, the policy configuration follows the SLAs established between the infrastructure owner and the two MVNOs considered in this scenario (blue and red).

Exploiting the tidal effect in C-RAN

In a mobile network, there are traffic variations throughout the day depending on whether the cell site is located in a residential or office area. These traffic variations create the so called *tidal effect* [6]. In a C-RAN architecture the shared pool of BBU resources can be efficiently used by exploiting the tidal effect. To model this behavior, the 7 cell sites in the emulation are split in two categories. First, there are sites that correspond to office areas (sites 1-4), and second, there are sites that correspond to residential areas (sites 5-7). A typical traffic pattern that illustrates the tidal effect is shown in Figure 6.3 [6]. It can be noticed in the figure, that the cell sites for the office areas have a higher load during the working hours, while the sites in the residential areas have more traffic load during the evening. Some cell sites have a high load at a particular moment during the day, requiring stringent limitations with respect to the network capacity. Alternatively, for other sites the load is low, so that very loose limitations need to be enforced to share the capacity. By following the traffic patterns, the infrastructure owner should be able to define policies for how the fronthaul network capacity is shared among different cell sites and MVNO's. It is also desirable that the policies are fine grained in order to follow the traffic load changes throughout the day (e.g. on an hourly basis). This not only eases the management operations for infrastructure sharing, but also leads to a better utilization of the network resources, given that the resources reallocation occurs on short time intervals. The traffic pattern illustrated in Figure 6.3 and described above is used in the following to present the details of the proposed service, with respect to capacity sharing policies.

Service details

The proposed service shares the network capacity between the MVNOs by rate limiting the traffic per cell site and per MVNO according to high level policies. To achieve this, two policies are created, each of them reflecting the SLA between the infrastructure owner and one MVNO (blue or red). A policy contains multiple rule entries, where one entry rate limits the traffic to a host in the emulated network. Hence, a rule entry contains the following elements: an identifier, the source and destination IP addresses for the traffic flow to be limited, the time interval when the rule applies, and the percentage of the link bandwidth allocated to the traffic flow. Since there is only one host per MVNO per site, the allocated bandwidth in a rule entry represents

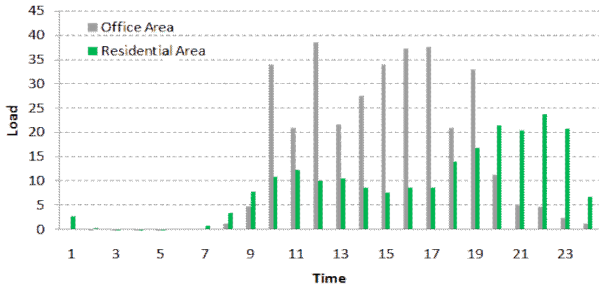


Figure 6.3: Typical traffic load variations throughout the day in a mobile network

the maximum allowed bandwidth for the corresponding MVNO for that site. The current service model does not consider the excess bandwidth or more complex QoS mechanisms. Rule entries in the policies are defined with a granularity of one hour, with respect to the time interval they are enabled for. This is a reasonable design decision which can easily capture the traffic fluctuations throughout the day without being overly fine granular.

Policies for various MVNOs must be consistent in the infrastructure, even if they reflect agreements with individual MVNOs. Specifically, when several policies overlap, by being applied in the same time interval and for different MVNOs, it is necessary that the sum of the guaranteed bandwidth for these policies to not exceed the maximum link capacity for that cell site (i.e. RRH). This is however a simplistic assumption since, it is possible that in real C-RAN deployments the network capacity is oversubscribed, if additional knowledge about the traffic patterns is exploited by the infrastructure owner. In the service model presented here, there is no oversubscription of the network capacity. Table 6.1 shows a capacity sharing strategy that is employed for the seven cell scenario described above, and for the two MVNOs (A and B). There are entries in the table which suggest that there is no rate limiting applied to the traffic (i.e. Not Limited). This corresponds to the case where the capacity sharing service is not implemented, and MVNOs compete for the link bandwidth in the fronthaul, in a best effort manner. Additionally, for rule with index 7, the sum for the guaranteed bandwidth is 80% of the total link capacity, instead of 100%. This is because the QoS mechanism employed in the emulated network is based on three types of priority queues, with fixed configuration for the guaranteed bandwidth: 2, 4 and 8 Mbps. Given that the links are dimensioned to 10 Mbps, the percentages reflect the strategy in Table 6.1.

In large and complex networks, it is important to automate the QoS management operations in order to facilitate QoS provisioning. In the service model presented here, the policies are automatically enforced depending on the time of the day, so that after setting the high level policies, the infrastructure owner does not need to be involved

			Guaranteed capacity %	
Index	Site ID	Time interval (hour)	MVNO A	MVNO B
1	1 - 3	00 – 09	$N.L.^*$	$N.L.^*$
2	1 - 3	09 – 13	80	80
3	1 - 3	13 – 17	80	20
4	1 - 3	17 – 00	20	80
5	4	00 – 09	$N.L.^*$	$N.L.^*$
6	4	09 – 17	80	80
7	4	17 – 00	40	40
8	5 - 7	00 – 10	$N.L.^*$	$N.L.^*$
9	5 - 7	10 – 14	80	80
10	5 - 7	14 – 21	80	20
11	5 - 7	21 – 00	20	80

Table 6.1: Example of policy definitions [*N.L. = Not Limited]

in continuous management operations. Figure 6.4 shows the process flow diagram for applying the capacity sharing policies by using the proposed service. Hence, the diagram does not cover other operations like modification or deletion of policies.

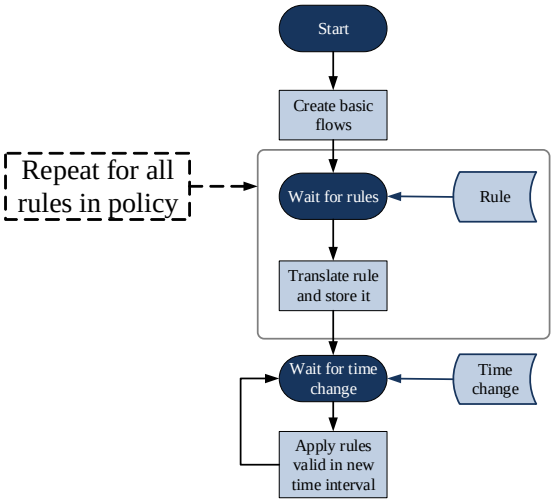


Figure 6.4: Flowchart for service behavior. The service automatically enforces policy rules for network capacity sharing, depending on the time of the day.

After the service starts, a set of basic OF entries are configured in the forwarding

devices (Figure 6.4), to enable forwarding of traffic flows between the the hosts and the Internet in the emulated network. This basic forwarding does not enforce any rate limiting upon the traffic flows. In the next stage, the policies are given as input to the service engine, which takes all the rule entries one by one and translates them into configuration commands. After all the policies and rules have been processed, the service moves into a new state, waiting for time changes (Figure 6.4). When a time change occurs, the service engine evaluates what rules are applicable for the new time interval and configures OF entries into the forwarding devices to enforce those rules.

6.1.3 Prototype implementation

A prototype for the capacity sharing service has been implemented over the Floodlight controller [15] and tested in the network model described above (lower part of Figure 6.2), which is emulated by using Mininet [23]. Various parts of the implementation and testbed are presented in the following.

Controller logic

An overview of the SDNC architecture including the proposed service is depicted in Figure 6.5. The architecture contains the typical Floodlight components as presented in previous chapters. For simplicity, only the *OF* Floodlight module is depicted, which realizes the communication with the forwarding devices in the emulated network. The underlying emulated overlay network consisting of forwarding devices is not shown, however, it is illustrated in Figure 6.2. The service logic, as presented in diagram 6.4, is implemented in the *Policy based infrastructure sharing* module. This module exposes a REST API (termed *Policy API* in here), which is used by administrators to configure policies for capacity sharing in C-RAN. Moreover, the module maintains an internal test timer variable, guiding which rules must be applied at a particular moment. For this, the module exposes another REST API (termed *Time API*) through which an external entity can change the internal test timer. The entire procedure involving policy and time configuration is explained in section 6.1.4.

Network model

For the capacity sharing service proposed here, it is feasible to consider the UEs only at the IP layer, hence they are emulated as Mininet hosts and identified by IP addresses. Each host is connected to an OF forwarding device, which corresponds to an RRH site. The site capacity, thus the capacity of the fronthaul link connecting the site to the BBU pool, is dimensioned to 10 Mbps. A typical LTE cell with a 20 MHz available spectrum can support a data rate (bandwidth) of up to 100 Mbps. As a result, the emulated network is dimensioned to approximately 10% of this bandwidth, in order to avoid performance issues in the virtualized environment (i.e. Mininet).

As mentioned, priority queues are used to rate limit the traffic flows in order to ensure correct capacity sharing of the frontaul link towards each RRH. Hence, there are

queues installed at the output ports in the OF overlay network (Figure 6.2) so that the traffic flows corresponding to various hosts is directed through these queues according to the policies that are applied in the network. Policies must be consistent before being applied in the network. This consistency verification is currently performed manually when policies are defined, prior to being processed by the proposed capacity sharing service.

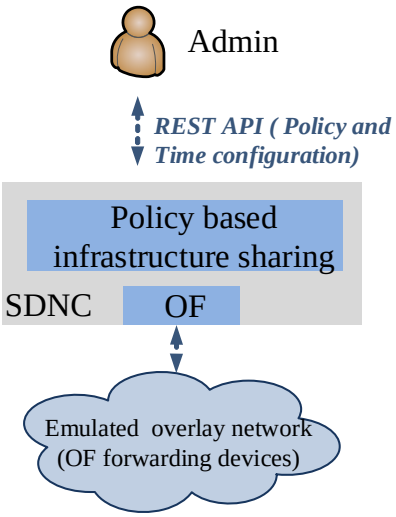


Figure 6.5: SDNC architecture. The emulated overlay network is depicted in the lower part of Figure 6.2

6.1.4 Evaluation and results

To demonstrate the practicality and the advantages of applying policies and automation for network management, the prototype is evaluated, and the outcomes are presented in this subsection. A simple methodology is used for the evaluation. First, a set of policies for the two MVNOs is installed in the network by using the *Policy API* (Figure 6.5). These policies implement the capacity sharing strategy illustrated in Table 6.1. Next, a test script uses the *Time API* to change the internal timer in the Policy based infrastructure sharing module, thus emulating time changes throughout the day. According to these time changes, various capacity sharing rules are enforced in the network, limiting the bandwidth available to each MVNO at each RRH site. To measure the bandwidth utilization, each host transmits a TCP traffic flow with a rate of 10 Mbps towards the server host (H-0, Figure 6.2). In this case, the throughput reported by the traffic generation tool (Iperf [19]) reflects the capacity sharing policies that

are enforced at a particular moment of the day. Hence, by observing the throughput variations, it is possible to infer the policy enforcement in the network.

For the first test, the internal service timer is set to hour 00 (midnight), by using the *Time API*. In this case there is no rate limiting applied (entries 1, 5 and 8 in Table 6.1), resembling a situation where the capacity sharing service is not deployed. Figure 6.6 shows the throughput measurements for each cell site. Since no rate limiting policies are enforced, the two MVNOs must compete at each cell site for the available capacity in a best effort manner. As it can be noticed, each MVNO gets approximately half of the entire capacity, however, registering large fluctuations. In some cases (e.g. site 7) one MVNO gets significantly more bandwidth that the other. Given that the network links are dimensioned to 10 Mbps, the maximum achievable throughput for a cell site is approximately 8Mbps, due to the TCP congestion control algorithm, and to the fact that average values are plotted. Throughout the experiment, the cumulated bandwidth utilization for each cell site does not exceed this 8Mbps threshold.

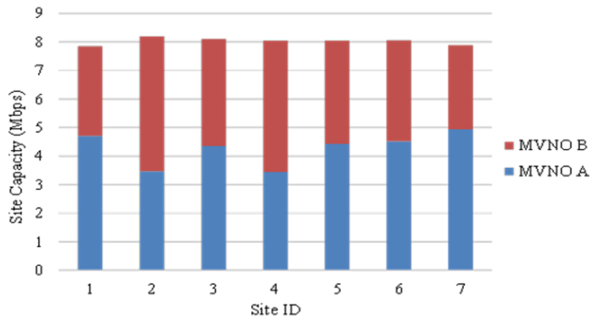


Figure 6.6: Capacity sharing when no policies are enforced in the network

The traffic mixture in the network can be controlled when the service is used and policies are enforced. For the next test case, the internal timer is set to hour 10 (Table 6.1), which corresponds to allocating 80% of the capacity to each MVNO at every cell site. In this case the network capacity allocation to various MVNOs at each site follows a predictable pattern, resulting in each MVNO receiving 50% of the link capacity. Figure 6.7 presents the measurements performed for this test case. Moreover, this case employs oversubscription of resources since more bandwidth is promised to both MVNOs than what the actual network links can support (i.e. 10 Mbps). Specifically, a bandwidth of 8 Mbps is allocated to each MVNO, summing up to 16 Mbps which is almost twice than the actual link capacity. Nonetheless, even with this loose capacity sharing constraint (i.e. oversubscription), the bandwidth is allocated in a much more predictable fashion than for the previous test case. Moreover, with oversubscription of resources, an MVNO can use more that 50% of the link capacity when the other MVNO is using less that 4Mbps.

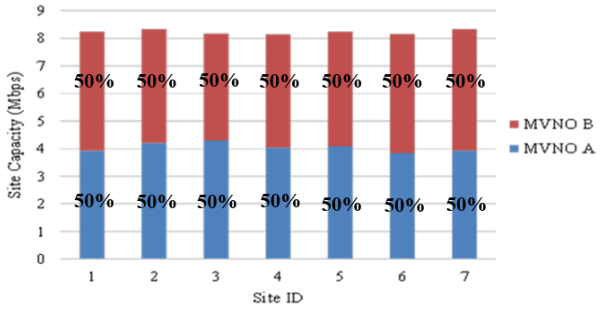


Figure 6.7: Capacity sharing when policies for equal resource allocation are enforced in the network

In the last test, the internal timer is configured to hour 22 (Table 6.1). According to the sharing strategy presented in the table, the infrastructure owner allocates 20% of the capacity to MVNO A and 80% to MVNO B for the first three cell sites (1-3). For cell site 4, each MVNO gets 40% of the capacity, by using rate limiting queues with 4 Mbps guaranteed bandwidth. Lastly, for the cell sites 5-7, the capacity sharing is mirroring the policies in site 1-3. As the results in Figure 6.8 confirm, the sharing is accurately enforced in the network. Even if the MVNOs transmit traffic flows with a 10 Mbps rate each, it is possible by configuring policies to rate limit the traffic to 6 Mbps for both MVNOs (as for site 4).

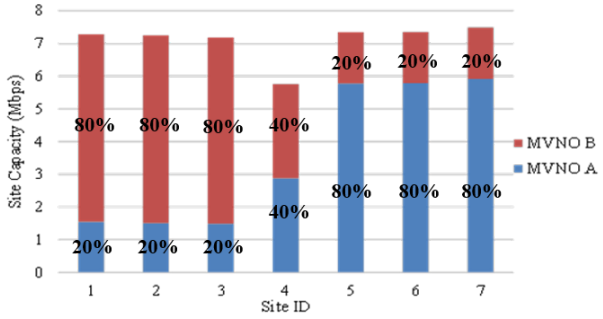


Figure 6.8: Capacity sharing at each cell site when various policies are enforced in the network

6.1.5 Summary

An important requirement for C-RAN is the high capacity needed in the fronthaul. A possible answer to this requirement is sharing the network infrastructure between several MVNOs. In this way, instead of having dedicated network resources, each MVNO uses a share of the same physical network, thus reducing the cost for deploying and operating a dedicated network. This sharing must consider various network conditions in order to be efficient, such as the type of cell site (office or residential) or the time of the day, both of which influence the traffic patterns in the network.

A policy based architecture is necessary for managing the capacity sharing mechanism, in order to be flexible enough to follow the daily traffic patters. This section proposes an SDN based solution for the capacity sharing challenge. To illustrate the proposed solution, a network model for the C-RAN scenario was defined and presented here. On top of this network model, the capacity sharing service was implemented, which takes as input a set of policies and enforces these policies automatically, depending on the time of the day. Thus, there is no need for human intervention in order to manage the capacity sharing service in the network. This simplifies greatly the management operations and leads to an increased adoption of proper QoS mechanisms instead of network overprovisioning techniques.

The results presented in the section show that when the service is not enforcing any particular policies for capacity sharing, the MVNOs compete for the available bandwidth in a best-effort fashion. However, the allocated resources can be easily controlled by changing the time of the day, which leads to enforcing certain policies for capacity sharing.

6.2 A generic policy framework for SDN

Networks are becoming increasingly complex and more difficult to manage. A specific use case to prove this trend has been introduced in the previous section. Moreover, it has been also demonstrated that a policy based approach to managing networks greatly simplifies the operations. While in the previous section the C-RAN scenario was used to demonstrate the applicability of SDN for policy based management, in this section the focus is on Data Centers (DCs). First, a short analysis introduces what are the challenges in DCs, and then a possible approach towards answering the challenges is presented.

The *de facto* standard for delivering IT functionality in enterprise areas and not only is through cloud services. Cloud services assume that a set of IT and network resources are provisioned in a pay-as-you-go business model. Usually, these underlying resources reside in Data Centers, and their coordination and flexible provisioning leads to stringent requirements from the DC architecture.

A typical DC architecture is depicted in Figure 6.9. The architecture is split in three logical planes. First, the data plane contains the actual IT and network resources

(e.g. compute servers, network links, etc.). Second, the control plane usually contains the decision logic for controlling various aspects of the data plane resources. This logic is split in independent entities, named controllers, depending on the type of resource to be managed: compute controller, network controller, etc. Third, there is the management and orchestration plane which is simply termed *orchestration plane* in here. Some of the functions belonging to the orchestration plane are: management of resources, customers and provisioning of services (e.g. dashboards). Automation is included in the orchestration plane, allowing human operators to perform complex task much easier. For example, coordinated provisioning of IT and network resources is easily performed if the orchestration plane is automated compared to configuring individual resources controllers. Finally, over the DC architecture there are entities that usually interact or make use of the DC services such as administrators, customers and applications (Figure 6.9).

Data Centers (DCs) continue to evolve to keep up with these increasing requirements in terms of agility, capacity, and QoS, leading to overly complex architectures. This complexity stems from having multiple functional entities (e.g. routing, orchestration, monitoring, etc.), each of them exposing its own functionality. Some areas in which DCs are being improved and drive the complexity of the architecture are:

- Optical elements are being adopted in the data center networks, enabling all optical or hybrid packet-circuit DC networks (DCNs). This leads to reduced congestion in the network, further improving the QoS [9] [36].
- Automation of operations at the control and orchestration planes level. This can reduce the service provisioning time, which impacts the QoS for the customers of cloud services.
- Adding advanced features at the Application Programming Interface (API) level (Figure 6.9), for provisioning of cloud services such as support for QoS automatic scaling (up or down) of the IT and network resources [34], [35].
- Adding advanced joint orchestration of network and IT resources to improve the overall resource utilization.

A common solution to simplify the management of complex systems is to apply policy based techniques [42]. There are several research works in the area of policy based management,especially for network management. The aim of these solutions is to simplify QoS configuration, network monitoring and service provisioning [29] [30] [32]. However, the previous works focus mostly on resources configuration and service provisioning. Managing the ever increasing complexity of the control and orchestration planes in DCs has not been consider in prior works. Without simple means to manage the complexity of the entire DC architecture, it becomes impossible for the architecture to be flexible and follow the changing business and customer

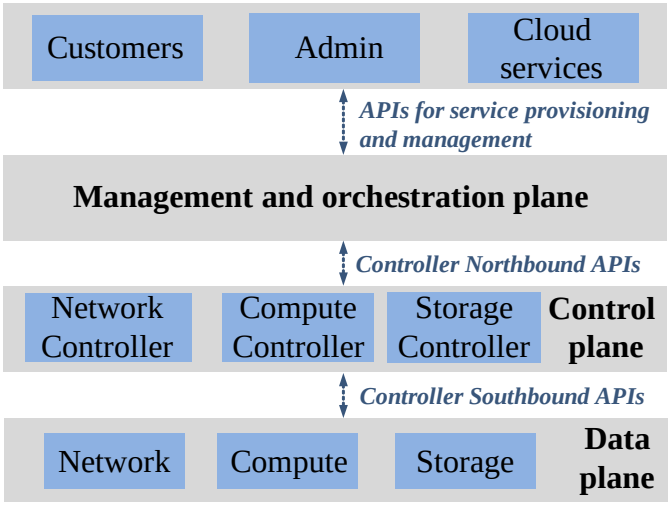


Figure 6.9: Generic data center architecture including all the three layers: data plane, control plane, management and orchestration plane

requirements. To this end, the work presented in this section introduces a generic framework for managing all the functionality at the data, control and orchestration planes in DCs. The core strengths of the policy based framework proposed here are:

- A unified mechanism to create and enforce policies, through standardized information models.
- The framework is extensible by employing a loosely coupled and model-driven architecture.

To illustrate the applicability of the policy framework, a DC use case is described herein. Additionally, a prototype has been implemented and a series of tests have been executed to validate it and to illustrate its performance.

The remaining of the section is organized as follows. The background on policy based management is given in subsection 6.2.1. The proposed policy framework is described in subsection 6.2.2 and the use case is presented in 6.2.3. The prototype is evaluated and the results are discussed in subsection 6.2.4, following that subsection 6.2.5 concludes this work.

6.2.1 Background and related work

Policy based management in SDN

The SDN paradigm is used to a great extent as a network control plane solution for today's DCs, opening up for new network management mechanisms. Due to the decoupling of control and data planes promoted by SDN, it is possible to control the network's behavior from a single entity, the SDN Controller (SDNC). Additionally, SDN allows for programmability of the control logic. These are inherent characteristics of policy based frameworks, opening up the possibility for combining SDN and policies for network management.

Significant research work has been devoted to Policy Based Network Management (PBNM) [39]. These previous research works resulted in various standards, innovative research oriented approaches and also solutions that are deployed in real networks. A core aim of the prior research in PBNM is to simplify QoS configuration, enabling QoS provisioning instead of just network overprovisioning [26] [3] [43]. Several PBNM concepts have been reused in the SDN architecture, resulting in a variety of SDN based policy frameworks being proposed in the recent years [29] [30] [13] [32] [28]. Most of these frameworks are discussed in the following, however, without completely exhausting the list of existing solutions.

The first category of works in PBNM deals with low level policies that are targeted for specific areas. As an example, [4] presents a reactive framework for policy based QoS provisioning, which enforces a set of QoS policies in the network, monitors the network and adapts the control plane rules to avoid policy violations. Another similar work, but with a broader scope is [13]. In this paper, the authors propose an API that allows network administrators to configure policies for access control, bandwidth allocation and also to control the paths in the network. Additionally, through the API, administrators can query the status of the network and also give hints to the SDNC about the future traffic patterns.

A second type of research works in SDN based network management leverages high level policies for expressing the network's behavior [30] [33] [27] [28]. These high level policies are termed *intents* in the literature [30] [27]. Network intents are used to capture administrators' intention regarding the behavior of the network, thus these intents introduce a declarative approach to policies. Through network intents an administrator provisions services by defining the characteristics of the service instead of how it should be implemented or configured in the network. On the other hand, [33] and [28] take an application centric approach. In this approach, policies are used to describe the demanded network infrastructure according certain application needs. These types of policies and network intents simplify the service provisioning in communications networks, especially in DCs, where they are largely deployed nowadays. Generally, these high level policy based approaches move the complexity from manually configuring the data plane elements into the automated decision logic at the SDNC.

Several works raise the level of network abstraction and propose specific programming languages to manage various aspects of the network such as forwarding behavior, access control, QoS, and others [22] [16] [18] [12]. Unlike the network intents and policy solutions discussed in the previous section, these works focus their effort in defining the language itself (e.g. semantics, grammar etc.), proving its correctness and utility in a number of scenarios. [18] defines a declarative language for specifying policies for network access control, QoS and Network Address Translation. Similarly, in [16] the authors propose Frenetic, a language which not only allows for defining policies, but also enables composition of high level programs for network control. Hence, the administrator can apply multiple programs such as monitoring, routing, on the same traffic in a sequential or parallel fashion. In [12], NetIDE is proposed as a development environment for SDN applications which allows for porting an application to any underlying controller. NetIDE is based on a common representation format (termed interchange representation format) of the application program, which is then translated to any underlying controller through a specialized driver. Yet another effort in defining a common network modeling language is NEMO, which is driven by standardization bodies and resulted also in an open source project [29] [44]. NEMO defines a language, inspired from Structured Query Language (SQL), for applications and administrator to configure intents for the network. The NEMO open source project implements a complete framework for network modelling APIs, based on the OpenDaylight (ODL) platform [31].

All the research works presented so far propose solutions for policy based resource management and service provisioning. Network intents, high level policies and policy specification languages open the possibility for human operators to simply declare the wanted network behavior instead of low level programming of traffic flows. However, SDN control planes are growing in complexity by containing more and more functionality. In this case, another challenge that must be addressed is managing all this functionality. To this end, existing PBNM solutions must be enhanced to become broader in scope, by providing means to control every aspect of the network.

Policy based management in DCs

Apart from the networking part, policy frameworks that are used in DCs must also consider other aspects such as the IT resources. Joint orchestration and provisioning of both IT and network are critical features for offering complex cloud services. In this direction, the individual resource controllers (i.e. compute, network, storage) expose APIs for resource management [35] [33]. These APIs can be considered as policy based mechanisms for services provisioning. Moreover, there are initiatives to built orchestration platforms on top of the functionality provided by the resources controllers [34]. Such orchestration platforms are based on templates that capture a complete service description (all the needed resources and their behavior under certain conditions), which are then translated and enforced by the framework by using

the individual resource controller APIs.

Recently, the open source community started the Openstack Congress project [32]. Congress aims to define a language and framework for declarative policy based management in DCs. While the works presented above are targeted towards specific areas, Congress aims to manage every aspect of a DC. As for example, by using Congress one can configure various functional entities, services and resources. Congress leverages database tables to implement the framework for enforcing the policies. These tables capture various *data sources*, which serve as input for the policy enforcement logic. A data source can be for instance monitoring information or other information generated by control or orchestration plane components. The information provided through the data sources, is used by the policy enforcement engine in Congress to validate policies, verify policy conflicts, and enforce the corresponding actions. The actions are enforced by using the features provided by other DC components, through their exposed APIs. For example, an action to allocate more bandwidth to a network connection is enforced by conveying this requirement to the network controller.

The work presented in this section takes a similar approach as Openstack Congress: it creates a middleware for defining and enforcing policies. The difference is that instead of using database tables to enable the communication between the DC components, the framework proposed here leverages a model driven architecture, offering better flexibility and extensibility when compared to Congress. While significant effort is put on defining the policy language in Congress, here, the policy definition is not based on a specific language. Alternatively, the focus in this work is on reusing standardized information models, allowing for better interoperability among different policy processing entities.

6.2.2 Policy framework

Policy information model

As previously mentioned, a standardized IM is used here to define policies. The IM is described in [26] and [3]. From the complete IM presented in the standards, only the core features have been reused and implemented. These features are graphically illustrated in the diagram in Figure 6.10 and the rest of the section will describe only the implemented features. A policy is termed *PolicyRule* in the IM, and it contains two main subcomponents: conditions (represented by the *PolicyCondition* interface) and actions (represented by the *PolicyAction* interface).

PolicyCondition has two specific implementations: *SimplePolicyCondition* and *CompoundPolicyCondition*. *SimplePolicyCondition* is the basic building block for defining conditions. A condition denotes when a policy is valid, so that if the condition is *True*, then the *PolicyRule* can be enforced. *CompoundPolicyCondition* allows for composing *SimplePolicyCondition* objects in Conjunctive or Disjunctive Normal Form sentences (CNF or DNF). In this case, if the CNF/DNF sentence holds *True*, then the *PolicyRule* is

valid and its actions are enforced. At its root, every condition is built from *PolicyVariables* and *PolicyValues*. A variable-value association indicates that a variable must be equal to the indicated value in order for the condition to hold *True*. As an example, a *PolicyVariable* can be time of the day (hour) and the associated *PolicyValue* can be a 20. In this example, the *PolicyRule* is enforced when the time of the day is 20 (i.e. 8 PM).

Likewise, *PolicyAction* is defined in a similar way as the *PolicyCondition*. However, *CompoundPolicyAction* is a simple set of *SimplePolicyAction* objects that are enforced in a specific order. When a *SimplePolicyAction* is enforced, the variable indicated by the *PolicyVariable* object must be made equal to the value contained in the *PolicyValue*. For instance, enforcing an action can translate in a configuration command that the link bandwidth must have 50 Mbps, which is communicated to the network controller and further enforced in the data plane.

This mechanism for defining policies is very similar to [32], in which database tables have been used to hold the variables and actions and to communicate the data between the DC components. Conversely, in here the IM is implemented as Java objects in a model driven fashion, being easily extensible by adding more types of variables into the IM.

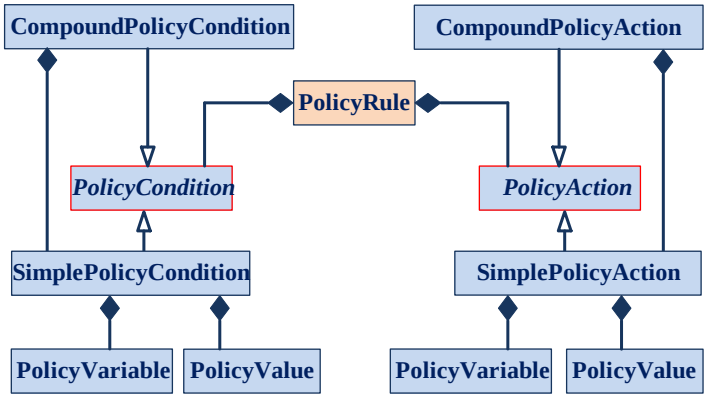


Figure 6.10: Policy Information Model

Policy state machine

The lifecycle of a policy is denoted through a state machine, which is illustrated in Figure 6.11. Every policy starts by being in the *Uninstalled* state, denoting that the administrator did not enforce the policy. When the administrator enforces a policy, its state moves into the *Installed*, which is a temporary state. In this temporary state, the policy conditions are evaluated and the conflict validation against other installed policies is performed. If the conditions are *True* and there are no conflicts, the policy

is enabled and it moves into the *Installed and enabled* state. Further, the actions comprised by the policy are applied. It is also possible that the conditions are not *True*, or there are existing conflicts. In this case, the policy moves into the *Installed and Disabled* state. In this state, the actions are not applied and the policy is in a stand by state waiting for the actions to become *True*. Moreover, each policy has a priority value associated with it. Thus, a policy can move from enabled to disabled by being overwritten by another conflicting policy which has a higher priority.

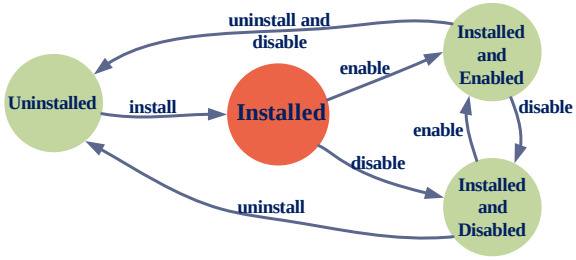


Figure 6.11: State machine for policies

Policy framework architecture

The architecture of the proposed policy framework is shown in Figure 6.12. The OpendDaylight (ODL) controller platform provides the basis for the framework [31]. Two characteristics of ODL are very important for the policy framework. First, ODL has a modular architecture, where each module implements a specific functionality, and second, ODL provides the means for modules to communicate with each other through its internal *data store*. Each ODL module can keep its data in a tree structure inside the data store, and also access data that other modules are keeping in the data store. Thus, the data store enables a uniform and well defined mechanism for communication between the ODL modules. This characteristic is exploited by the policy framework to create a middleware for gathering data from other modules, use the data for the policy enforcement process, and then communicate actions to other modules.

The YANG modeling language [24] is used to define the tree structures inside the data store for keeping the data generated by the ODL modules. Given that the communication between the modules is governed by YANG models, the architecture is model-driven. The functionality and the extensibility of the framework is driven by models, which are backwards compatible. This is more flexible than the database based solution used in Congress [32].

As also illustrated in the figure below, the data store has three partitions depending on the type of data that is kept. The *Policy Repository* partition is used to keep uninstalled policies. From this repository, an administrator can choose to install

certain policies at a given moment. The second partition is termed *Installed Policies*, and contains installed policies that are either enabled or disabled. The differentiation between enabled and disabled policies in this partition is realized by an attribute inside the policies. When a policy is installed and also enabled, its actions must be enforced. For this, the actions are put in the third partition, termed *Policy Actions*. In this case, a module targeted by a policy reads the actions of that policy from this partitions and enforces them accordingly.

It is important to mention that the partitions in the data store can be accessed not only by ODL modules, but also by external entities through an automatically generated API. Hence, the framework can be used to manage the ODL based SDN control plane, and also other entities in the DC architecture, becoming a DC wide policy based management solution.

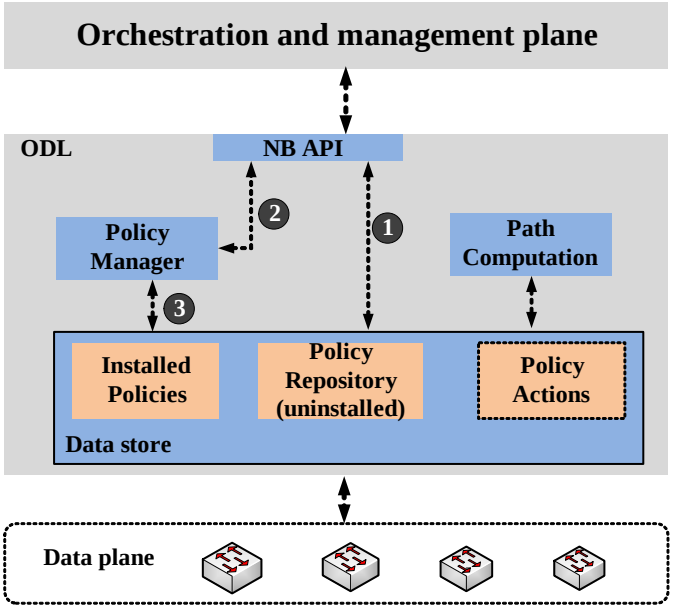


Figure 6.12: Architecture of the policy framework

The Policy Manager (PM) module is the entity which implements the policy processing logic. This logic comprises policy validation, conflict verification and communicating the policy actions to other modules by writing them in the correct partition in the data store. More details regarding the complete lifecycle of a policy are given in the next paragraph.

Policies can be of several types, depending on the type of variables contained in their conditions and actions. For example, there can be a policy to control the

algorithm of the Path Computation (PC) module (Figure 6.12). Enforcing such a policy implies that a set of variables are configured in the *Policy Actions* data store partition. Further, the PC module reads the new values for the associated variables and adjusts itself to reflect the new configuration.

Policy enforcement workflow

Policies can be configured by using the Northbound API (NB API), exposed by the PM module (Figure 6.12). This API comprises two types of operations. First, there are CRUD (create, read, update, delete) operations (denoted with 1 in the figure). By using thes operations, one can add new policies to the *Policy Repository* or remove and modify existing ones. To be added to the *Policy Repository*, policies are first defined in a JSON format [41], and then added by using the create operation of the NB API. The second type of operations are *installation*/*uninstallation* (denoted with 2 in the figure). An administrator identifies a policy that exists in the *Policy Repository* and installs it by using its unique ID. This installation request is first received by the PM, which retrieves the entire policy from the *Policy Repository*. Then, PM verifies if the policy conflicts with an existing installed policy. If there are any conflicts, it enables the new policy only if it has a higher priority than the conflicting ones. This process finishes once the corresponding policy actions are written in the *Policy Actions* partition in order to be enforced by the target module. Finally, the policy is written in the *Installed Policies* partition Conversely, if the policy cannot be enabled due to invalid conditions or existing conflicts, the policy is still written in the *Installed Policies* partition, but is left in the *Installed and disabled* state. No actions are generated and enforced for policies in this state.

The policy framework has two modes of operation, with respect to policy enforcement: (1) a polling mode, and (2) a subscribe/notify mode. In the polling mode, the target modules (e.g. PC) poll the *Policy Actions* partition for policy changes and then fetch and apply their corresponding policy actions. In the subscribe/notify mode, each module first subscribes for changes in the *Policy Actions* partition in the data store. Once a new policy is enabled, the corresponding module is notified about the event and given the changed data. The module receives the changed policy and applies the actions by self–configuration, according the policy variables.

The proposed policy framework is an infrastructure (middleware) for defining and validating policies, and communicating policy actions to corresponding modules. Further, the target modules adjust themselves according to the policy actions. Even though the policy framework is based on an existing SDN controller (i.e. ODL), it can be used for policy-based management of the entire DC and not only for the network, as it will be exemplified in the following.

6.2.3 Use case

The use case described here is based on the architecture depicted in Figure 6.13. In the data plane there are compute entities (i.e. servers) interconnected through a hybrid, electrical packet switched and optical circuit switched network. The paths in the network are computed by the PC module within ODL, which also hosts the PM, as previously explained. At the control plane level, the DCN Optimization (DCNO) component operates as a separate entity from ODL, continuously monitoring the network and computing the best circuit configuration to achieve increased network throughput. The computed circuit configuration is passed to ODL which further sets up the wanted circuits in the network. Continuing to the orchestration level, the Resource Orchestration Algorithms (ROA) module performs joint compute and network resource orchestration to provision cloud services. Specifically, it receives service requests from customers and, according to certain business rules and SLAs, it maps the requested service resources into the physical data plane resources.

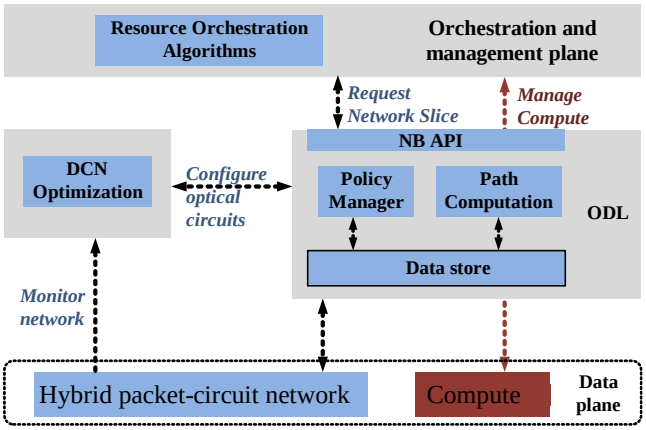


Figure 6.13: DC architecture to exemplify the use case

In the context described above, two YANG models have been defined to create the policies for the ROA and DCNO modules. After compiling the YANG models, ODL creates the data store trees to keep these two types of policies, and the actions of the policies. Additionally, the NB APIs are also generated, allowing the external modules and administrators to communicate with the ODL-based policy framework. To have a complete functionality, the PM is enhanced with methods for conflict validation for these two policies (DCNO and ROA). Since the ROA module contains an integer linear programming algorithm for resource allocation, the policy defined for this module contains the set of variables needed to configure the algorithm. On the other hand, the DCNO module contains algorithms that decide when and what circuits must be established in the network to offer better QoS to certain traffic flows. For this, it fetches

the network topology and monitoring data about the traffic in the network. Using this data, it identifies the long-lived and high bandwidth traffic flows (i.e. elephant flows) and it computes the feasibility of assigning dedicated optical circuits to offload these flows. An administrator can control the behavior of the algorithm (thresholds, timers, etc.) by configuring policies specially defined for this purpose, through the NB API.

In both cases (ROA and DCNO), the policies may have empty conditions (that are always true), indicating that the administrator wishes to enforce a particular configuration regardless of the existing policies. In the use case presented here, which is part of a prototype implementation, the policy framework is used to perform configuration management.

6.2.4 Evaluation and results

The aim of this evaluation is to illustrate the behavior and assess the performance of the proposed policy framework in various scenarios. For this, the following test scenario is used. Three policy files are defined, each of them containing 29, 58 and 100 unique policy rules, respectively. A file contains policies of 5 different types: DCNO, ROA, PC policies and two other types of policies that are used for controlling the link bandwidth. All policies are added to the *Policy Repository* by using the NB API. Further, all the policies are installed one by one through the NB API. Because there are conflicts among policies, not all of them reach the *Installed and Enabled* state.

As a performance metric for the above scenario, the Round Trip Time (RTT) to successfully enable policies is used. Hence, policies that are installed, but due to conflicts cannot be enabled, are not included in the assessment. To measure the RTT, the following process was used. When being installed one by one, the individual RTT for the policies is measured. First, an installation request is issued by a script and the current time is recorded (i.e. time A). Other scripts emulated the target modules, by subscribing for changes in the *Policy Actions* data store partition. When a policy installation finishes with successfully enabling the policy, the corresponding actions are written to the *Policy Actions* partition and the target script is notified. Continuing, the script records the time of receiving the notification (i.e. time B). By subtracting the two time values (time B - time A), the RTT is computed. This performance assessment includes also the RTT for the notification between the data store and the scripts that emulate the target modules. However, this bias exists in all the results and it does not affect the conclusions made in this section.

Figure 6.14 shows the measured instantaneous RTT (noted as installation time in the figure) for the three different sets of policies. While, in general, the measured RTT is approximately 20 ms, there are noticeable spikes in the RTT throughout the test, in all three cases. These spikes are correlated with the cases in which the policy to be installed conflicts with several already installed policies. In consequence, if the priority of the policy to be installed is higher than for the conflicting policies, the PM must execute significantly more operations. These operations are to remove the

conflicting policies from the data store, and to write the new policy. This need for multiple operations, especially multiple data store accesses, creates the spikes in the RTT measurements. To emphasize, there is a correlation between the RTT and the number of conflicting policies. Hence, for better performance, it is advisable to design a solution that decreases the number of possible conflicts between the policies.

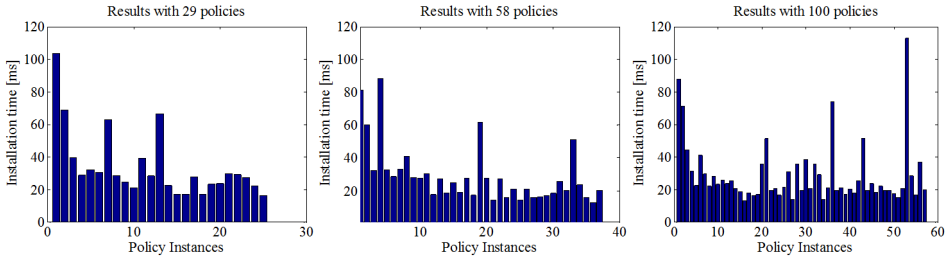


Figure 6.14: Instantaneous RTT measurements for the 29, 58 and 100 policies sets respectively

Disregarding the spikes mentioned above, the measured RTT is not increasing throughout the test. This means that there is no correlation between the number of installed policies and the time it takes to install a new policy. Therefore, the performance of the framework is not hindered by large sets of policies, unless they lead to multiple conflicts, results in conflicting policies that have to be overwritten by new policies.

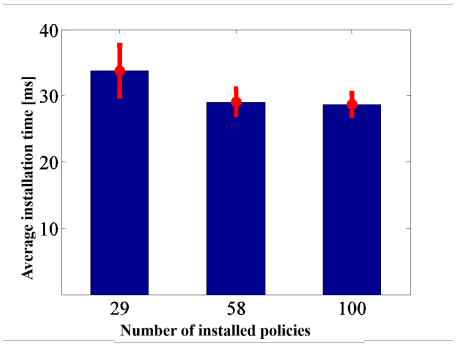


Figure 6.15: Average RTT for the three sets of policies, with standard deviation

The average RTT for each test case has been computed and plotted in Figure 6.15, including also the standard deviation. In this figure, there is a noticeable reduction in the average installation time (RTT) once the number of policies is increased from 29 to 58 and then 100. This pattern is attributed to the fact that the impact of the spikes

in the measurements is bigger when there are fewer policies, these spikes raising the overall average. In general, the results are stable and the performance is good with an average of 30 ms for the end-to-end installation time. However, the performance may be negatively affected if more complex policies are defined, that contain more variables and values

6.2.5 Summary

To simplify the management of QoS, many solutions propose policy based mechanisms. By using these type of mechanisms, administrators capture their intention regarding network's behavior in simple policies, following that the underlying policy framework automatically enforces these policies. Most of the existing policy frameworks focus on services and network resource configuration, essentially covering the NP parameters part of the QoS spectrum (bandwidth, delay). Nevertheless, managing the Non Network Performance (NNP) parameters configuration is also very important for provisioning of QoS and is usually disregarded in the existing policy frameworks. For example configuring the control and orchestration plane components in DCs has not been considered in the literature.

This section describes a potential solution to the challenge of managing every aspect of a complex architecture, taking as an example a typical DC environment. This makes the architecture very flexible in coping with changing business and customers requirements, eventually leading to better QoS (with respect to both NP and NNP). The proposed policy framework is based on a standardized IM and a model-driven software architecture, to allow for better interoperability and easy future enhancements. Overall, this section argues that every aspect of a network and IT architecture must be managed in a simple way, to reduce the human error in configuration and the service provisioning time. This eventually leads to better QoS for the services and reduced OPEX. In this context, SDN is seen as an enabling technology due to its inherent centralization and programmability, which are key aspects for an automated management framework.

6.3 Chapter summary

The work presented in this chapter concentrates around two core ideas. First, policy based management techniques are essential in enabling better resource utilization and simpler management operations. This idea was emphasized in section 6.1 through a practical, real world example from mobile networks domain: C-RAN requires significantly more capacity in the fronthaul part of the mobile network in order to transport the raw data from the antennas. This requirement can hinder the adoption of C-RAN, which has many other benefits when compared to current mobile radio access technologies (explained in the section). By intelligently sharing physical network infrastructures, mobile operators can overcome this potential limitation with respect to

network capacity. The section further argued that SDN is a feasible solution in enabling intelligent sharing of the network, since it brings programmability to networking and also centralization of control plane logic. Programmability is the key requirement for an automated system, because through programmability, manual configuration operations can be written in intelligent autonomous programs. Additionally, through centralization, an autonomous program can gather enough information about the network in order to take optimal decisions about the network's behavior (e.g. resource allocation, services, routes, etc.). Hence, the work described in this section proposed an SDN based autonomous framework for capacity sharing. At the core of the framework there is a policy based mechanism, through which administrators can express their high level intentions, according to various SLAs.

The second important idea presented in this chapter is that existing frameworks for policy based network management do not consider a critical element: control and management plane configuration management. With the increasing complexity of autonomous control plane logic, network architectures become very rigid and hard to manage. Thus, apart from service provisioning and resources configuration management, policy frameworks must also consider configuration management of the control and management planes. That is, an administrator must be able to outline the strategy for the control and management plane logic in a simple way, through policies. For this, policy based management frameworks must broaden their scope. Section 6.2 argued about this idea, and presented the work that supports the argument through a generic policy based management solution. Unlike the first section which was focused on SDN, this second section concentrated on generic policy based management, taking the DC environment as an example for this. However, SDN is still a key concept in policy based management, according to the argument brought by the first section.

CHAPTER 7

Traffic optimization in data center networks

The previous chapter demonstrated the applicability of SDN for simplifying QoS provisioning through policy based management. SDN introduces a high degree of programmability into the networking domain, allowing for network automation and thus facilitating more and more intelligent policy based management frameworks. The second important feature brought by SDN is centralization of control plane logic. This feature is critical for improving QoS in communication networks because it allows for global optimization algorithms. Through global optimization, resource utilization can be improved and resource allocation among different traffic types can be performed in a fair fashion. Essentially, the centralized control logic in SDN eliminates the need to execute complex distributed protocols in order to obtain and spread knowledge in the network.

This chapter aims to leverage the inherent centralization in SDN to implement intelligent algorithms for traffic optimization in order to achieve better traffic differentiation, aiming to improve QoS. A Data Center (DC) environment is used for the work in this chapter. In this environment, a novel network architecture is introduced, on top of which the algorithms are executed. After, the algorithms are assessed by using various topologies and traffic patterns in order to quantify the improvement they introduce with respect to network resource utilization.

7.1 Introduction and motivation

Existing Data Centre Networks (DCNs) continue to evolve to keep up with the stringent application requirements in terms of bandwidth, latency, agility, etc. Traditional DCNs are based on hierarchical, tree-like topologies with three layers: Top of the Rack(ToR) switches, aggregation switches, and core switches at the root of the tree. An important drawback of such topologies is the over-subscription of the links towards the core switches, which often leads to congestion. To cope with this, various enhancements are proposed in the literature, ranging from Layer 0, applying new technologies (e.g. optical switches) and designing new architectures (e.g. fat-tree, hypercube, etc.), up to Layer 3 (complex traffic metering and forwarding schemes) [6].

In the data plane, flatter network topologies with higher node degree, such as fat-

tree and hypercube, increase the overall network performance (e.g. higher bisection bandwidth, higher resiliency to link failures). Another trend in DCNs is replacing electrical packet switching in the core with optical circuit switching. This immediately improves network performance parameters including latency and bandwidth.

Several existing works combine flat topologies with optical circuit switching to obtain better performance for the DCNs [5] [14]. Figure 7.1 depicts a typical hybrid flat network topology for DCs. In this topology, there are two logical planes: Electrical Packet Switched (EPS) plane and the Optical Circuit Switched (OCS) plane. The EPS plane contains the ToR switches while the OCS plane comprises one or more Optical Circuits Switches (OCSs).

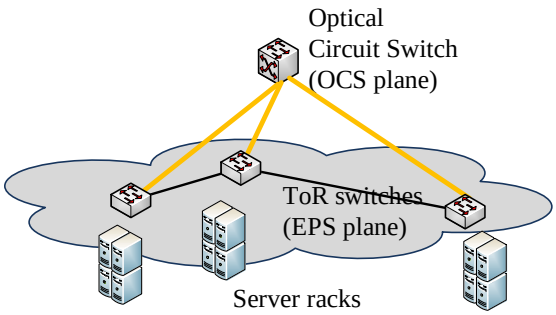


Figure 7.1: Simplified hybrid electrical packet switched/optical circuit switched network architecture. It contains one layer of ToR switches interconnected among themselves and one layer of OCSs that offer on demand circuit based interconnectivity between the ToR switches.

Traffic flows in the DCN are usually forwarded through the EPS plane. To obtain better performance (e.g. lower delay, more bandwidth), existing solutions employ algorithms that selectively offload traffic from the EPS plane by using optical circuits in the OCS plane [5] [14]. Two types of traffic flows are usually differentiated in these algorithms. First, there are the short-lived and fast varying flows termed *mice flows*, and second, there are the long-lived traffic flows that require significantly more bandwidth, termed *elephant flows*. Existing algorithms in the literature perform detection of elephant traffic flows in the EPS plane and offload them through optical circuits. However, these algorithms have some limitations that constrain the maximum performance of the DCN in terms of supported traffic throughput. After these limitations are described in the next section, an SDN based architecture is proposed, together with a novel algorithm for offloading traffic from the EPS to the OCS plane. The algorithm is implemented and tested in two DCN topologies to prove that it brings several benefits when compared to existing solutions. Throughout the chapter, the terms *circuit* and *optical circuit* are used interchangeably.

7.2 Exploiting hybrid network architectures

The DCN topologies considered in here consist of a single layer of ToR switches (in the EPS plane) and one high radix optical circuit switch (at the OCS plane). All the network devices are OF enabled. Specifically, the ToR switches are typical electrical packet switched OF forwarding devices, and the OCS is also OF enabled (allows for optical circuit configuration through OF optical extensions). As mentioned above, the key aspect of hybrid DCN topologies is the algorithm for traffic optimization. Generally, the algorithm should route part of the traffic that is transmitted over the EPS plane, through optical circuits in the OCS plane. In this way the congestion in the EPS plane can be reduced, while the offloaded traffic experiences lower delays and higher throughput due to optical links.

Existing algorithms employ elephant flow detection at the ToR switches or at the virtual switches inside the servers that connect to the ToR switches. Based on this, elephant flows are routed over optical circuits. However, current solutions [5] [14] [10] utilize optical circuits only for the elephant traffic flows that originate and end at the ToR switches at the circuit endpoints. This type of optical circuit is termed *private circuit* here. When private circuits are employed, other elephant traffic flows that are transmitted between the circuits endpoints, but do not originate and end at the these endpoints, are not transmitted through the circuit. As a result, these type of elephant flows are still transmitted through the EPS plane, leading to congestion and non trivial delay in the ToR switches. This concept is also illustrated in Figure 7.2. The optical circuit S2-S8-S3 is a private circuit, therefore, the elephant traffic flow transmitted by S1 to S3 is not routed over the circuit. Instead, the elephant flow utilizes the more scarce network resources in the EPS plane, leading to degraded quality for the mice flows that are also being transmitted in the EPS plane.

To alleviate the limitations imposed by private circuits in current solutions, the current work proposes a DCN architecture which inherently enables sharing of the optical circuit by elephant flows that are not generated or ended at the circuit endpoints. This type of optical circuit is termed *shared circuit* here. When a shared circuit is created between two endpoints (e.g. ToR switches), all the elephant traffic flows that are transmitted between those endpoints are routed over the circuit, regardless of where the flows have originated or of their destination. Consequently, the load on the EPS is reduced leading to better quality for the mice flows and better differentiation among mice and elephant flows.

The concept of shared circuit is illustrated in Figure 7.2. S4-S8-s6 is a shared circuit, hence, the elephant traffic flow sent from S4 to S7 is routed over the circuit on the S4-S6 segment. As a result, on the S4-S6 segment in the EPS plane, the mice flows can experience better quality due to lower traffic load.

An important aspect when applying OF based SDN for traffic optimization is the footprint of OF entries in the forwarding devices. For every optical circuit, the circuit must first be configured in the corresponding OCS. Then, the elephant traffic flows are

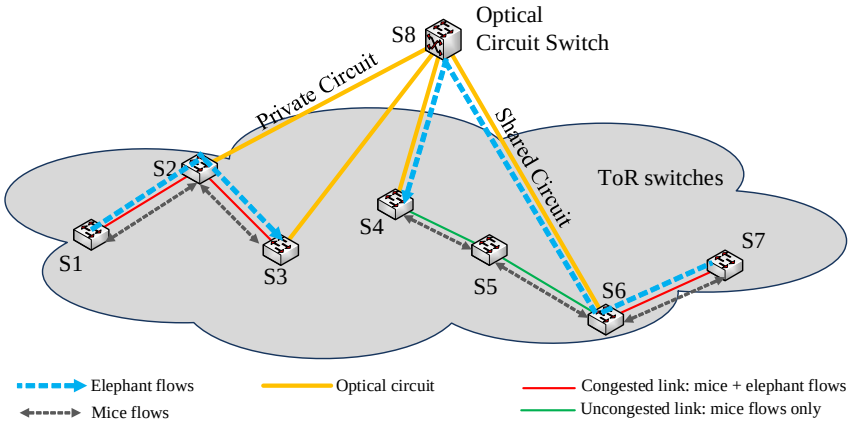
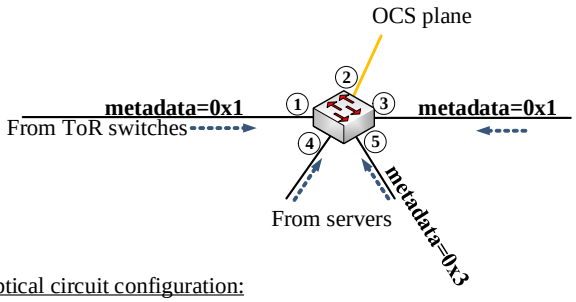


Figure 7.2: Private versus shared optical circuits in hybrid DCNs

routed to the circuit by installing a set of OF entries in the forwarding devices (ToR switches) at the circuit endpoints. However, this mechanism of having a set of OF entries per elephant traffic flow leads to a large number of OF entries being installed in ToR switches (i.e. OF forwarding devices). Hence, ToR switches can have their flow table space exhausted and not be able to accommodate new traffic flows. To cope with this, the solution introduced in this chapter employs a mechanism to reduce the OF entries footprint in the ToR switches.

By exploiting the metadata based matching in OF version 1.3 [11], the solution for shared optical circuits presented here can be implemented in an efficient way, with respect to the number of required OF entries in the ToR switches. Figure 7.3 shows a specific example of OF forwarding entries configuration for a ToR switch. Each ToR switch has an initial configuration for OF entries that adds a certain metadata field to all the packets incoming at every port in the switch. Depending on the port type, this metadata is different. Specifically, ports that connect to the servers transport data that is associated with private circuits, so that packets arriving at these ports are tagged with a given metadata (Figure 7.3). On the other hand, ports that connect to ToR switches are tagged with another metadata. For shared circuits, both metadata described above must match so that the traffic from both types of ports is routed through the optical circuit. The OF entries that are required in order to create private or shared optical circuits are shown in Figure 7.3. According to these rules, a private circuit takes only traffic from ports 4 and 5 in the figure, while a shared circuit takes traffic from ports 1, 3, 4 and 5. In consequence, instead of installing per flow OF entries, by employing the metadata mechanism proposed here, one OF entry per circuit is enough to route all the traffic flows through the circuit. This significantly

reduces the number of OF entries in the ToR switches, easily solving the challenge of accommodating a large number of traffic flows in the DC.



Private optical circuit configuration:
table=1,ip,**metadata=0x3/0x2**, nw_tos=252,nw_dst=10.0.0.0/24, actions=output:2

Shared optical circuit configuration:
table=1,ip,**metadata=0x3/0x1**, nw_tos=252,nw_dst=10.0.0.0/24, actions=output:2

Figure 7.3: OF entries configuration for private and shared circuits

7.3 DC architecture

On top of the hybrid network topology described previously, there is a complete DC architecture, which comprises various control and management plane functions. An overview of the entire DC architecture is given in Figure 7.4. The data plane contains servers, ToR switches and OCSs. Elephant traffic flows are detected in the Open vSwitch (OVS), inside the servers, by using traffic sampling based on the sFlow protocol [12]. The sampled traffic data is received by the *Elephant Flow Detector* (EFD) service (step 1), which upon detecting a new elephant traffic flow, marks the packets belonging to that flow with a certain Differentiated Services Code Point (DSCP) value (step 2). Further, the marked packets are transmitted into the DCN to reach their destination (step 3).

The *Network Observer* (NO) module uses sFlow to monitor the status of the elephant flows at the ToR switches (step 4). Based on the monitoring data, NO generates information for the *Network Scheduler* (NS) module (step 5), regarding the elephant flows and the overall traffic load observed in the EPS and OCS planes. By using this information, NS decides which circuits to configure, and sends the corresponding configuration commands to the SDN Controller (SDNC) (step 6). The SDNC, which keeps the logical topology of the DC, implements the commands received from NS, by installing OF entries in the data plane. Specifically, the SDNC configures optical

circuits in the OCS and routes elephant traffic flows to these circuits by installing OF entries in the ToR switches (step 7).

To take efficient decisions regarding the optical circuits to be configured, the NO and NS modules require detailed information such as complete network topology, types of links and devices. This information is generated in step 0 in the figure, and it is communicated to the NS and NO modules. For example the network topology is retrieved by the NS from the SDNC.

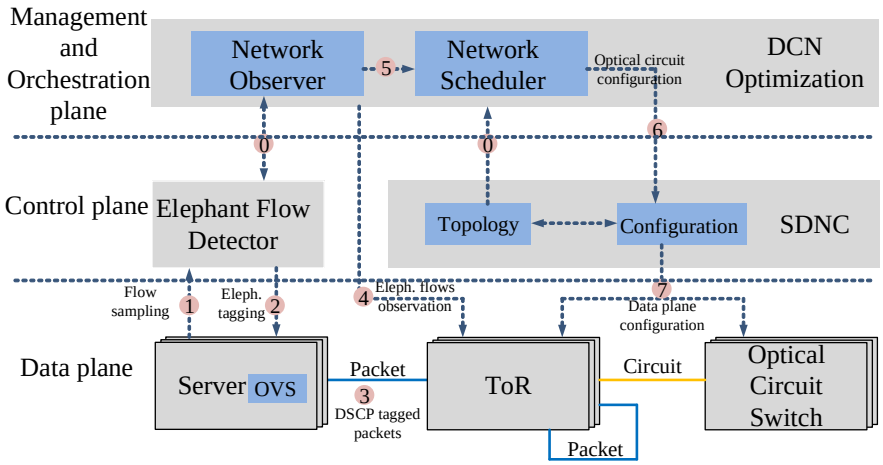


Figure 7.4: Complete DC architecture for the hybrid EPS/OCS network

7.4 Optimization algorithms

Optical circuits are configured based on the status of the EPS and OCS planes in the network (e.g. traffic flows, load, etc.). NO and NS modules implement the algorithms for gathering the status information, processing this information, and selecting the optimal circuits to be configured. First, NO implements the algorithm presented in 4, which is mainly targeted at gathering and processing the information about the observed traffic flows at the ToR switches.

There are two databases in algorithm 4: an electrical database (EDb) and an optical database (ODb) (lines 1-2). EDb contains information about the traffic flows that are transmitted over the EPS while ODb contains information about the elephant traffic flows that are routed over the OCS plane. Both databases are continuously updated with sFlow monitoring information from the ToR switches. At the same time, the execution of the circuit selection algorithm, which is implemented in the NS module, is triggered with a given rate (line 3, and lines 20-23). By default,

`executeCircuitSelection()` is executed every 100 ms. Overall, algorithm 4 is a wrapper for gathering information and triggering the execution of the circuit selection algorithm, which is explained next.

Algorithm 4 Network Observer generic algorithm

```
1:  o_db ← database for optical plane
2:  e_db ← database for electrical plane
3:  rate ← rate of optical selection algorithm
4:  while True do
5:      s ← next sample from sFlow collector
6:  update databases:
7:      f_ID ← flow ID from s
8:      tp ← ToR pair for s
9:      if f_ID in e_db then
10:         e_db.updateDescription(f_ID)
11:     else if f_ID in o_db then
12:         o_db.updateCircuit(f_ID)
13:     else
14:         flow ← create flow description from s
15:         if o_db.hasCircuitFor(tp) then
16:             o_db.addFlow(f_ID, flow)
17:         else
18:             e_db.addFlow(f_ID, flow)
19: selection algorithm:
20:     if current_time - previous_time > rate then
21:         executeCircuitSelection()
22:         previous_time ← current_time
```

Algorithm 5 is implemented in the NS module, and it describes the logic for the selection of optical circuits. Since the algorithm is relatively complex, only a generic description of it is given here, while the details are captured in the pseudocode. In general, based on the updated information in the two databases (i.e.EDb and ODb), the algorithm selects circuits iteratively, by using a simple heuristic. A *ToR pair* in the algorithm is a tuple of ToR switches that are the endpoints of a traffic flow. Hence, each traffic flow is characterized by a ToR pair, which comprises, on one hand, the switch where the traffic flow is detected, and on the other hand, the switch at the other end of the traffic flow.

The circuit selection algorithm maintains the priority queue data structure (line 3), which is used to select the optical circuits. Essentially, the algorithm examines all the elephant traffic flows in the database and creates a list of all the ToR pairs for these elephant flows. Each ToR pair has a given weight. Moreover, the ToR pair and

Algorithm 5 Optical Circuit Selection

```
o_db ← mapping for circuits and elephant flows
e_db ← mapping for ToR pairs and traffic flows
pq ← max Priority Queue
cc ← optical circuit conflicts
W1 ← weight threshold for circuit
W2 ← weight threshold for conflict preemption
D ← duration threshold for conflict preemption
for tp in e_db do
    w ← computeWeight(tp)
    pq.push(tp, w)
Found ← False
while pq.hasMore() and not Found do
    bp ← pq.pop()
    if weight(bp) > W1 then
        if not o_db.hasConflicts(bp) then
            configureCircuit(bp)
            update o_db
            Found ← True
        else
            preempt ← True
            for cp in o_db.getConflictPairs() do
                if weight(cp) < W2 then
                    cc.updateDuration(cp)
                else
                    preempt ← False
                d ← cc.getDuration(cp)
                if d < D then
                    preempt ← False
            if preempt then
                for cp in o_db.getConflictPairs() do
                    removeCircuit(cp)
                configureCircuit(bp)
                update o_db
                Found ← True
```

its corresponding weight are pushed as a tuple into the priority queue. The weight is computed to directly reflect the total throughput for the elephant traffic flows for that ToR pair (i.e. transmitted between the ToR switches). ToR pairs usually have several associated elephant flows. Next, the algorithm selects the "best" ToR pair from the priority queue, which outputs the tuple (i.e. pair + weight) with the highest weight. Therefore, the algorithm chooses as candidate for optical circuit, in a greedy fashion, the ToR pair with the highest throughput for the elephant traffic flows. When the best selected ToR pair has a weight that is higher than W1 (line 5), then the ToR pair is actually considered for being offloaded over an optical circuit. If this is the case, a circuit is configured between the ToR switches denoted by that ToR pair.

An important feature of the algorithm is that it allows for circuits to be preempted. It may be the case that a ToR pair, which is a good candidate for being offloaded to optical circuit, conflicts with another existing circuit. This constraint is raised because in the current architecture there is only one optical link for each ToR switch, connecting it to the OCS. Knowing this, a ToR switch cannot be involved in two optical circuits simultaneously. In case there is a conflict, the algorithm allows that the newly discovered candidate ToR pair preempts the existing conflicting circuit, given that certain conditions are met. One condition is that the weight of the existing circuit is lower than W2, which is usually configured to be even lower than W1. A second condition is that the first condition is valid for a certain duration (denoted as D, in line 7). This mechanism gives the chance to new ToR pairs to be offloaded to optical circuits, if they have a higher weight for a given time duration. The algorithm implements this logic by using another data structure denoted as `cc` (line 4). Finally, if preemption can be done (conditions are met), the existing optical circuit is removed, and a new circuit is configured for the candidate ToR pair. Essentially, this particular mechanism tries to avoid two suboptimal cases. First, it avoids old circuits being still maintained, albeit only transporting a low amount of traffic. Second, it avoids oscillations in optical circuits reconfiguration.

7.5 Evaluation and results

To prove the feasibility of the proposed idea, a series of tests have been performed, and the results are presented in this section. The methodology used for the tests was to have two topologies for the EPS plane (ToR switches). Further two traffic patterns have been tested. In consequence, there have been 4 tests performed, resulting from the combination of the chosen topologies and traffic patterns. For these 4 tests, both private and shared circuit strategies have been employed, to assess the performance improvement brought by applying shares optical circuits instead of the private ones.

Mininet is used to emulate the data plane. Hence, the entire network (i.e. EPS + OCS) is emulated by using OVS devices. While emulating the electrical packet switched ToR switches is straightforward by using OVS devices, the OCS devices

require extra features. To emulate a circuit switched device (i.e. OCS), the typical OVS instance is abstracted at the controller level to allow only port-to-port connectivity. Moreover, one port can be interconnected with one other port only at a given time, thus resembling a optical circuit switched device, which switches the light signal arriving at one port to another port, without combining these light signals. To suit an emulated environment, the links in the data plane have been dimensioned to 10 Mbps for the packet switched links and 100 Mbps for the emulated optical circuits. Iperf [7] was used to generate the traffic flows in the emulated DCN.

Control and management plane components of the DC architecture are implemented by using Python scripts. The SDNC is also emulated through Python scripts and by leveraging configuration utilities for OVS.

Test topologies

A flat DCN architecture is considered here for the EPS plane (comprising the ToR switches). Regarding the OCS plane, there is a single circuit switch that provides on demand optical circuits to the ToRs. In the following, the OCS is omitted from the topology and also from the corresponding figures.

For the tests, two particular flat EPS topologies are chosen. The first topology is a ring of 10 ToR switches, which offers simple connectivity (Figure 7.5). This topology has been used by Facebook [4] and Google [13] in their data centers to interconnect server clusters.

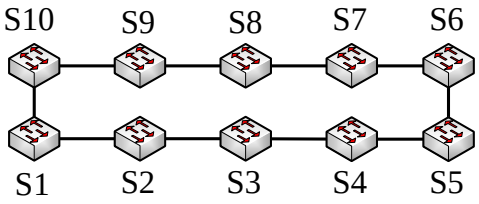


Figure 7.5: Ring topology with 10 ToR switches

The second topology is a Flattened Butterfly (FBFly) which leverages high radix switches to create a scalable network, with a low diameter leading to reduced delay. Specifically, the topology comprises 9 ToR switches that are interconnected as a 3-ary-3-flat topology [9]. Figure 7.6 illustrates the interconnections for the first group of 3 ToR switches only (outlined in blue color). The other two groups of three ToR switches are connected in the same fashion, creating the complete 3-ary-3-flat topology.

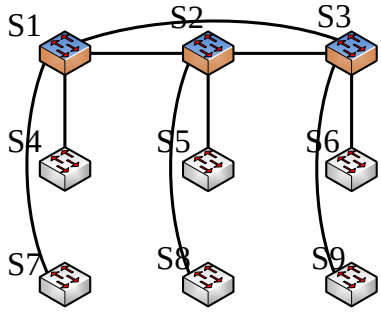


Figure 7.6: Flattened Butterfly topology 3-ary-3-flat

Traffic traces

Network traffic traces from University of Wisconsin (UNI1), reported in [2], have been used here for the evaluation. From these traces, the TCP session characteristics are extracted and all the sessions above 32 KB are considered as elephant flows and marked with the corresponding DSCP in the network. Also, the time intervals between the session are reduced in order to obtain a moderate network load.

A second traffic trace (termed *uniform* in the test) was created based in the traffic characteristics reported in [3] [1] [10] [8], which state that 10% of the traffic flows are elephant flows and they transport 90% of the data. Specifically, traffic flows were generated with a random distribution of sessions for mice flows (2 KB to 32 KB), and for elephant flows (32 KB to 100 MB), complying with the criteria stated above. Moreover, the sources and destinations of the traffic flows are uniformly distributed among the nodes in the network showing no particular clustering for the flows. After the trace was created, the same trace was used for all the tests performed.

Results

Figures 7.7 and 7.8 present traffic throughput comparisons, as reported by Iperf, for the different topologies and traffic traces used here. Mice and elephant flows are considered separately, since the proposed architecture promotes a better split between these types of traffic flows in the DCN. As it can be noticed in the results, shared circuits improve the throughput of the traffic flows in every test case, when compared to private circuits.

In general, the results show a higher throughput for the UNI1 traffic trace than for the *uniform* trace, even though both traces have a similar traffic load. This observation can be attributed to the fact that UNI1 has more skewed traffic, which is better exploited through shared circuits. Specifically, UNI1 presents more clustered traffic patterns such that multiple elephant flows from different sources share the same

destination. Hence, shared circuits further improve the performance of the network when the traffic presents these skewed patterns.

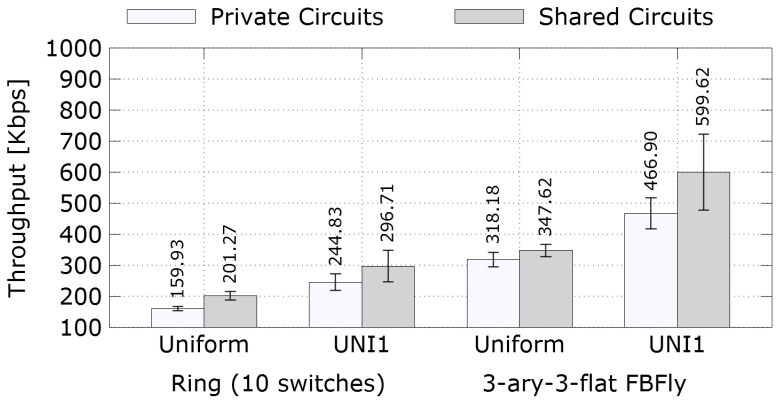


Figure 7.7: Average mice throughput

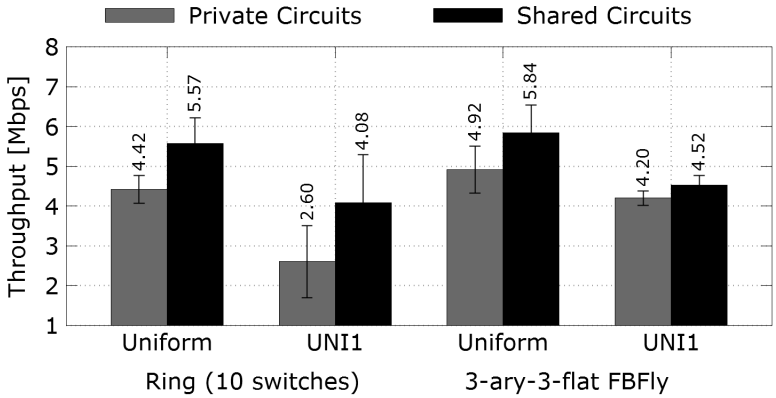


Figure 7.8: Average elephant throughput

Ring topologies have a lower node degree, making the devices in such a topology to be less well connected when compared to a Flattened Butterfly (FBFly) topology. Due to this, the results show a higher throughput for the 3-ary-3-flat FBFly topology. However, even under the constraint of having lower connectivity, the performance of the ring topology can be improved by employing shared circuits instead of private circuits. Moreover, the shared circuit strategy brings more performance improvements when deployed in the ring topology than in the FBFly topology. Specifically, while

the throughput in the ring topology is improving with 21% - 57% when using shared circuits instead of private circuits, in the FBFly is improving significantly less, 8% - 28%.

7.6 Chapter summary

An important characteristic of SDN is the centralization of the control plane decision logic in the network. This chapter argued that this characteristic can be used to increase the performance of the networks and the QoS of the network services. For this, a DC scenario was introduced here, in which both EPS and OCS technologies are applied for better network performance. The aim of this hybrid technology (EPS/OCS) is to offer better differentiation, thus QoS, between mice and elephant flows. Specifically, elephant flows can utilize the more resourceful optical links given that these flows require more bandwidth and are usually long-lived. At the same time mice flows can utilize the EPS which becomes less congested due to elephant flows being offloaded to optical. Through this traffic differentiation, both types of flows receive better QoS.

However, existing traffic optimization solutions for hybrid DCN topologies expose potential limitations in the way they use optical circuits. These limitation have been introduced in the chapter, and have been addressed through the proposed shared circuits strategy. The centralized SDN based DC architecture proposed in the chapter facilitates gathering network monitoring information from the DCN, implementing optimization algorithms for exploiting shared circuits, and configuring the DCN according to the algorithm decisions. Network monitoring is realized by using the sFlow protocol, while traffic forwarding and circuit configuration is implemented by using OF v1.3. The proposed solution introduces a novel mechanism for routing elephant traffic flows to optical circuits which significantly reduces the number of OF entries in the ToR switches. The mechanism applies per circuit OF entries instead of per traffic flow.

Results demonstrated that, by employing shared circuits, the throughput for both elephant and mice traffic flows is significantly improved. In particular, shared circuits better exploit skewed traffic patterns, and bring higher performance improvements in ring topologies.

The work presented in this chapter is based on the work presented in Abstract I and Paper J, which resulted from the author's collaboration with IBM Research Labs, Haifa.

CHAPTER 8

Conclusion and future outlook

This chapter aims to highlight the contributions of each previous chapter to answering the initial research question (stated in chapter 1). Furthermore, the challenges encountered in the research conducted for this thesis are emphasized here.

The initial research question is re-stated:

Can SDN be applied to make QoS provisioning more scalable, so that it can be deployed in large and complex networks? In particular, is QoS provisioning simplified and further improved by using SDN techniques?

To answer this research question, one should first seek to understand how network services are provisioned in SDN. To this end, chapters 2 and 3 gave a complete overview of the SDN paradigm and, in particular, on how QoS provisioning is performed in SDN. Chapter 3 focused, first, on how SDN changes the way data plane resources are managed, when compared to traditional networks. Second, the chapter demonstrated the simplicity of creating QoS aware services in SDN through well designed QoS configuration APIs. The QoS slicing application was introduced as an example to prove that SDN has the advantage, over traditional networks, of simplifying QoS provisioning through programmability.

Employing SDN based QoS provisioning in large networks requires that the architecture can be easily scaled up. It has been stated in the literature that SDN exhibits certain scalability related drawbacks. At the data plane level, the biggest challenge, in the author's opinion, is the limited space available in the flow tables of the forwarding devices. This confines the applicability of SDN only to small network domains. Chapter 4 proved that this challenge can be tackled through a simple, well known mechanism: traffic flow aggregation.

At the control plane level, the biggest obstacle identified by the author, towards achieving scalability in SDN, is limited controller performance when the network grows. Careful optimization of the control functions' operating parameters eliminates the SDN controller performance bottlenecks - the solution proposed by the author in Chapter 5. This is a novel idea in the field of SDN, mainly because it does not rely on controller external mechanisms for scalability, unlike existing vertical or horizontal scaling techniques. What is more, the proposed solution can be gradually implemented, because it can coexist with current widely deployed techniques.

By eliminating the two abovementioned critical challenges identified in the SDN architecture, it is possible, in author's view, to scale an SDN based QoS provisioning

architecture even to the level of very large and complex networks.

The failure of existing QoS mechanisms in traditional networks stands in the complexity required by these, in order to be deployed and operated. This shortcoming has led to further research in simplifying network management, and in particular, QoS configuration management. To this end, policy based management arose as viable solution. Due to its inherent programmability, SDN is the best technology to achieve policy based management, and therefore, to simplify network management operations. Chapter 6 validates the hypothesis stated above, by demonstrating that an SDN implementation for policy based management is able to eliminate human intervention, and further automate network management. Additionally, the chapter reinforces this idea by introducing a model driven policy framework for configuration management of the entire network ecosystem.

In this context, the existing complex QoS mechanisms can avoid their obsolescence by adopting the SDN paradigm to implement automated frameworks for policy based management.

Chapter 7 uses a novel hybrid EPS/OCS network topology and proves that SDN can be applied in this topology to implement intelligent algorithms for traffic optimization towards achieving increased network resources utilization. Furthermore, the algorithms allow for differentiation among different types of traffic, which is critical for obtaining predictable QoS. In conclusion, the inherent centralization of the control logic in SDN allows and promotes the use of global optimization algorithms, which results in better resource utilization, and improved QoS for network services.

Based on the work and the results presented in this thesis, the author argues that SDN can significantly facilitate QoS provisioning for large and complex networks. As a result, NSPs have clear reasons to move forward from overprovisioning the network with more resources, towards applying SDN based QoS mechanisms. It can already be observed, that well known companies, such as Google, successfully apply SDN to increase the performance of their large scale networks, and simplify operations through automation [1].

It is worth mentioning that the industrial and academic SDN space are advancing extremely fast. This represented a challenge for conducting the research captured in this thesis, as the number of standards (i.e. OF), open source projects (i.e. Floodlight) and publications has been constantly on a rise, making it very difficult to propose solutions that did not rely on obsolete results or technologies. As such, NSPs should consider the pace of SDN advancements, when choosing the particular SDN technologies to deploy in their infrastructure.

Combined Bibliography

References from Chapter 1

[1] ITU-T. *Definitions of terms related to Quality of Service*. Recommendation E800. Telecommunication Standardization Sector of ITU, Sept. 2008 (cit. on pp. 1, 2).

[2] M. Yuksel, K. K. Ramakrishnan, S. Kalyanaraman, J. D. Houle, and R. Sadhvani. "Quantifying Overprovisioning vs. Class-of-Service: Informing the Net Neutrality Debate". In: *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*. Aug. 2010, pp. 1–8 (cit. on p. 2).

References from Chapter 2

[1] *Chapter 5 of Roy Fielding's doctoral dissertation introducing REST*. `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`. Accessed: March 2016 (cit. on p. 8).

[2] *Floodlight official web site*. `http://www.projectfloodlight.org/floodlight/`. Accessed: March 2016 (cit. on p. 7).

[3] ONF. *OpenFlow Switch Specification, version 1.0.0.* Standard. Open Networking Foundation, 2009 (cit. on pp. 7, 9).

[4] ONF. *SDN Architecture*. Tech. rep. Open Networking Foundation, June 2014 (cit. on p. 7).

[5] *Open vSwitch official web site*. `http://openvswitch.org/`. Accessed: March 2016 (cit. on p. 10).

[6] *OpenDaylight official web site*. `http://www.opendaylight.org/`. Accessed: March 2016 (cit. on p. 7).

References from Chapter 3

[1] A. V. Akella and K. Xiong. "Quality of Service (QoS) guaranteed network resource allocation via software defined networking (SDN)". In: *12th International Conference on Dependable, Autonomic and Secure Computing*. Dalian, China, Aug. 2014, pp. 7–14 (cit. on pp. 15, 33).

[2] B. Davie and A. Charny and J.C.R. Bennett and K. Benson and J.Y. Le Boudec and W. Courtney and S. Davari and V. Firoiu and D. Stiliadis. *An Expedited Forwarding PHB (Per-Hop Behavior)*. RFC 3246. IETF, Mar. 2002. URL: http://www.rfc-base.org/txt/rfc-3246.txt (cit. on p. 16).

[3] B. Davie and B. Pfaff. *The Open vSwitch Database Management Protocol*. RFC 3246. IETF, Dec. 2013. URL: https://tools.ietf.org/html/rfc7047 (cit. on p. 18).

[4] S. Banerjee and K. Kannan. "Tag-In-Tag: Efficient flow table management in SDN switches". In: *10th International Conference on Network and Service Management (CNSM) and Workshop*. Nov. 2014, pp. 109–117 (cit. on p. 27).

[5] M. F. Bari, S. Chowdhury, R. Ahmed, and R. Boutaba. "PolicyCop: an autonomic QoS policy enforcement framework for software defined networks". In: *IEEE SDN for Future Networks and Services*. Trento, Italy, Nov. 2013, pp. 1–7 (cit. on pp. 14, 31).

[6] T. Benson, A. Akella, A. Shaikh, and S. Sahu. "CloudNaaS: a cloud networking platform for enterprise applications". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. Cascais, Portugal, Oct. 2011 (cit. on pp. 14, 15).

[7] A. Botta, A. Dainotti, and A. Pescapè. "A tool for the generation of realistic network workload for emerging networking scenarios". In: *Computer Networks* 56.15 (2012), pp. 3531–3547 (cit. on p. 37).

[8] I. Bueno, J. I. Aznar, E. Escalona, J. F. Riera, and J. A. Garcia-Espin. "An OpenNaaS based SDN framework for dynamic QoS control". In: *IEEE SDN for Future Networks and Services*. Trento, Italy, Nov. 2013, pp. 1–7 (cit. on p. 14).

[9] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz. "On the effect of forwarding table size on SDN network utilization". In: *IEEE INFOCOM*. Toronto, Canada, Apr. 2014, pp. 1734–1742 (cit. on p. 27).

[10] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp. "OpenQoS: an openflow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks". In: *Signal and Information Processing Association Annual Summit and Conference*. Hollywood, California, US, Dec. 2012, pp. 1–8 (cit. on p. 14).

[11] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. "Participatory networking: an API for application control of SDNs". In: *ACM SIGCOMM 2013*. Hong Kong, China, Aug. 2013, pp. 327–338 (cit. on pp. 14, 31).

[12] C. Filsfils and J. Evans. "Engineering a multiservice IP backbone to support tight SLAs". In: *Computer Networks: The International Journal of Computer and Telecommunications Networking* 40.1 (Sept. 2002), pp. 131–148 (cit. on p. 12).

[13] *Floodlight official web site*. http://www.projectfloodlight.org/floodlight/. Accessed: March 2016 (cit. on p. 18).

[14] J. Guo, L. Fangming, T. Haowen, L. Yingnan, L. Hai, and L. John. “Falloc: fair network bandwidth allocation in IaaS datacenters via a bargaining game approach”. In: *21st IEEE International Conference on Network Protocols (ICNP)*. Goettingen, Germany, Oct. 2013, pp. 1–10 (cit. on p. 14).

[15] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. “Achieving high utilization with software-driven WAN”. In: *ACM SIGCOMM 2013*. Hong Kong, China, Aug. 2013, pp. 15–26 (cit. on pp. 14, 27, 31).

[16] *Iperf official web site*. `http://software.es.net/iperf/`. Accessed: March 2016 (cit. on p. 24).

[17] A. S. Iyer, V. Mann, and N. R. Samineni. “SwitchReduce: Reducing switch state and controller involvement in OpenFlow networks”. In: *IFIP Networking Conference*. Brooklyn, NY, US, May 2013, pp. 1–9 (cit. on p. 27).

[18] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. “B4: Experience with a globally-deployed software defined WAN”. In: *ACM SIGCOMM 2013*. Hong Kong, China, Aug. 2013, pp. 3–14 (cit. on pp. 14, 15).

[19] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula. “Automated and Scalable QoS Control for Network Convergence”. In: *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. San Jose, California, US, 2010 (cit. on pp. 15, 31).

[20] B. Lantz, B. Heller, and N. McKeown. “A network in a laptop: rapid prototyping for software-defined networks”. In: *Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Monterey, CA, US, Oct. 2010 (cit. on pp. 15, 18).

[21] M. Mechtri, I. Houidi, W. Louati, and D. Zeghlache. “SDN for inter cloud networking”. In: *IEEE SDN for Future Networks and Services*. Trento, Italy, Nov. 2013, pp. 1–7 (cit. on p. 14).

[22] ONF. *OpenFlow Switch Specification, version 1.0.0.* Standard. Open Networking Foundation, 2009 (cit. on pp. 18, 27).

[23] ONF. *OpenFlow Switch Specification, version 1.1.0.* Standard. Open Networking Foundation, 2011 (cit. on p. 27).

[24] ONF. *OpenFlow Switch Specification, version 1.3.0.* Standard. Open Networking Foundation, 2012 (cit. on pp. 27, 29).

[25] *Open vSwitch manual*. `http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf`. Accessed: March 2016 (cit. on p. 33).

[26] *Open vSwitch official web site*. `http://openvswitch.org/`. Accessed: March 2016 (cit. on p. 18).

[27] *Opendaylight OVSDB project official web site*. https://wiki.opendaylight.org/view/OVSDB_Integration:Main/. Accessed: March 2016 (cit. on p. 32).

[28] D. Palma, J. Gonçalves, B. Sousa, L. Cordeiro, P. Simoes, S. Sharma, and D. Staessens. "The QueuePusher: Enabling Queue Management in OpenFlow". In: *Third European Workshop on Software Defined Networks (EWSDN)*. Budapest, Hungary, Sept. 2014, pp. 125–126 (cit. on pp. 31, 40).

[29] D. Qiang. "Network-as-a-service in software-defined networks for end-to-end QoS provisioning". In: *23rd Wireless and Optical Communication Conference (WOCC)*. Newark, NJ, US, May 2014, pp. 1–5 (cit. on p. 15).

[30] M. J. A. Qureshi and M. Saleem. "Simulation and Visualization of Transmission Control Protocol's (TCP) Flow-Control and Multi-Home Options". In: *2007 International Bhurban Conference on Applied Sciences Technology*. Jan. 2007, pp. 139–146 (cit. on p. 25).

[31] R. Enns and M. Bjorklund and J. Schoenwaelder and A. Bierman. *Network Configuration Protocol (NETCONF)*. RFC 6241. IETF, June 2011. URL: https://tools.ietf.org/html/rfc6241 (cit. on p. 18).

[32] A. R. Roy, M. F. Bari, M. F. Zhani, R. Ahmed, and R. Boutaba. "Design and management of DOT: A Distributed OpenFlow Testbed". In: *IEEE Network Operations and Management Symposium (NOMS)*. Krakow, Poland, May 2014, pp. 1–9 (cit. on p. 36).

[33] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss. *An Architecture for Differentiated Services*. RFC 2475. IETF, Dec. 1998. URL: http://www.rfc-base.org/txt/rfc-2475.txt (cit. on p. 12).

[34] Z. Wang and J. Crowcroft. "Quality-of-service routing for supporting multimedia applications". In: *IEEE Journal on Selected Areas in Communications*. Vol. 14. 7. IEEE, Sept. 1996, pp. 1228–1234 (cit. on p. 19).

References from Chapter 4

[1] Ceragon. *Wireless Backhaul Topologie: Analyzing Backhaul Topology Strategies*. http://www.winncom.com/images/stories/Ceragon_Wireless_Backhaul_Topologies_WP.pdf. Accessed: March 2016 (cit. on p. 53).

[2] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz. "On the effect of forwarding table size on SDN network utilization". In: *IEEE INFOCOM*. Toronto, Canada, Apr. 2014, pp. 1734–1742 (cit. on pp. 44, 45).

[3] J. Costa-Requena. “SDN integration in LTE mobile backhaul networks”. In: *The International Conference on Information Networking 2014 (ICOIN2014)*. Phuket, Thailand, Feb. 2014, pp. 264–269 (cit. on p. 48).

[4] S. Das, Y. Yiakoumis, G. Parulkar, N. Mckeown, P. Singh, D. Getachew, and P. D. Desai. “Application-Aware Aggregation and Traffic Engineering in a Converged Packet-Circuit Network”. In: *Optical Fiber Communication Conference and Exposition (OFC/NFOEC)*. Los Angeles, CA, 2011 (cit. on p. 47).

[5] *Floodlight official web site*. `http://www.projectfloodlight.org/floodlight/`. Accessed: March 2016 (cit. on p. 48).

[6] B. Lantz, B. Heller, and N. McKeown. “A network in a laptop: rapid prototyping for software-defined networks”. In: *Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Monterey, CA, US, Oct. 2010 (cit. on p. 53).

[7] B. Leng, L. Huang, X. Wang, H. Xu, and Y. Zhang. “A mechanism for reducing flow tables in software defined network”. In: *IEEE International Conference on Communications (ICC)*. London, UK, June 2015, pp. 5302–5307 (cit. on p. 47).

[8] S. Luo, H. Yu, and L. M. Li. “Fast incremental flow table aggregation in SDN”. In: *23rd International Conference on Computer Communication and Networks (ICCCN)*. Shanghai, China, Aug. 2014, pp. 1095–2055 (cit. on p. 47).

[9] R. Narayanan, S. Kotha, G. Lin, A. Khan, S. Rizvi, W. Javed, H. Khan, and S. A. Khayam. “Macroflows and Microflows: Enabling Rapid Network Innovation Through a Split SDN Data Plane”. In: *Proceedings of the 2012 European Workshop on Software Defined Networking*. EWSDN ’12. Darmstadt, Germany, 2012, pp. 79–84 (cit. on p. 45).

[10] A. Rasmussen, A. Kragelund, M. Berger, H. Wessing, and S. Ruepp. “TCAM-based high speed Longest prefix matching with fast incremental table updates”. In: *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*. July 2013, pp. 43–48 (cit. on p. 45).

[11] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. “PAST: Scalable Ethernet for Data Centers”. In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’12. Nice, France, 2012, pp. 49–60 (cit. on pp. 45, 46).

[12] R. Wang, D. Butnariu, and J. Rexford. “OpenFlow-based Server Load Balancing Gone Wild”. In: *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. Hot-ICE’11. Boston, MA, 2011, pp. 12–12 (cit. on p. 47).

References from Chapter 5

[1] *Amazon official web site*. https://aws.amazon.com/. Accessed: March 2016 (cit. on p. 61).

[2] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. "ONOS: Towards an Open, Distributed SDN OS". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA, 2014, pp. 1–6 (cit. on p. 60).

[3] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. "DevoFlow: Scaling Flow Management for High-performance Networks". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada, 2011, pp. 254–265 (cit. on p. 61).

[4] W. Dawoud, I. Takouna, and C. Meinel. "Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning". In: *Global Trends in Computing and Communication Systems: 4th International Conference, ObCom 2011, Vellore, TN, India, December 9-11, 2011. Proceedings, Part I*. Ed. by P. V. Krishna, M. R. Babu, and E. Ariwa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 11–25. ISBN: 978-3-642-29219-4 (cit. on p. 62).

[5] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. "Towards and Elastic Distributed SDN Controller". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 7–12 (cit. on pp. 59, 60).

[6] S. Dutta, S. Gera, A. Verma, and B. Viswanathan. "SmartScale: Automatic Application Scaling in Enterprise Clouds". In: *IEEE 5th International Conference on Cloud Computing (CLOUD)*. Honolulu, HI, US, June 2012, pp. 221–228 (cit. on p. 62).

[7] *Floodlight official web site*. http://www.projectfloodlight.org/floodlight/. Accessed: March 2016 (cit. on p. 64).

[8] S. Hassas Yeganeh and Y. Ganjali. "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 19–24 (cit. on p. 61).

[9] B. Heller, R. Sherwood, and N. McKeown. "The Controller Placement Problem". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland, 2012, pp. 7–12 (cit. on p. 59).

[10] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia. "Pareto-optimal resilient controller placement in SDN-based core networks". In: *Teletraffic Congress (ITC), 2013 25th International*. Shanghai, China, Sept. 2013, pp. 1–9 (cit. on p. 59).

[11] *Iperf official web site*. http://software.es.net/iperf/. Accessed: March 2016 (cit. on p. 68).

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. "Onix: a distributed control platform for large-scale production networks". In: *OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation*. Vancouver, Canada, Oct. 2010, pp. 351–364 (cit. on p. 60).

[13] A. Lenk and M. Turowski. "Vertical Scaling Capability of OpenStack - Survey of Guest Operating Systems, Hypervisors, and the Cloud Management Platform". In: *Service-Oriented Computing ? ICSOC 2014*. Lecture Notes in Computer Science. Springer International Publishing, Nov. 2014 (cit. on p. 61).

[14] *OFLOPS 2011*. http://archive.openflow.org/wk/images/3/3e/Manual.pdf. Accessed: March 2016 (cit. on pp. 75, 76).

[15] *Oracle VirtualBox official web site*. https://www.virtualbox.org/. Accessed: March 2016 (cit. on p. 69).

[16] K. Phemius and M. Bouet. "Monitoring latency with OpenFlow". In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. Zuric, Switzerland, Oct. 2013, pp. 122–125 (cit. on p. 65).

[17] A. Tootoonchian and Y. Ganjali. "HyperFlow: A distributed control plane for Openflow". In: *INM/WREN'10 Proceedings of the 2010 internet network management conference on Research on enterprise networking*. San Jose, CA , US, Apr. 2010 (cit. on p. 60).

[18] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. "On Controller Performance in Software-defined Networks". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. Hot-ICE'12. San Jose, CA, 2012, pp. 10–10 (cit. on p. 59).

[19] Z. Wang and J. Crowcroft. "Quality-of-service routing for supporting multimedia applications". In: *IEEE Journal on Selected Areas in Communications*. Vol. 14. 7. IEEE, Sept. 1996, pp. 1228–1234 (cit. on p. 67).

[20] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. "On scalability of Software-Defined Networking". In: *IEEE Communications Magazine* 51.2 (2013), pp. 136–141 (cit. on pp. 57, 59).

[21] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. "Scalable flow-based networking with DIFANE". In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2010) (cit. on p. 61).

[22] Y. Zhao, L. Iannone, and M. Riguidel. “On the performance of SDN controllers: A reality check”. In: *IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. San Francisco, CA US, Nov. 2015, pp. 79–85 (cit. on p. 75).

References from Chapter 6

[1] M. Y. Arslan, K. Sundaresan, and S. Rangarajan. “Software-defined networking in cellular radio access networks: potential and challenges”. In: *IEEE Communications Magazine* 53.1 (Jan. 2015), pp. 150–156 (cit. on p. 83).

[2] M. Artuso, A. Marcano, and H. Christiansen. “Cloudification of mmwave-based and packet-based fronthaul for future heterogeneous mobile networks”. In: *IEEE Wireless Communications* 22.5 (Oct. 2015), pp. 76–82 (cit. on p. 83).

[3] B. Moore. *Policy Core Information Model (PCIM) Extensions*. RFC 3460. IETF, Jan. 2003 (cit. on pp. 97, 99).

[4] M. F. Bari, S. Chowdhury, R. Ahmed, and R. Boutaba. “PolicyCop: an autonomic QoS policy enforcement framework for software defined networks”. In: *IEEE SDN for Future Networks and Services*. Trento, Italy, Nov. 2013, pp. 1–7 (cit. on p. 97).

[5] C. J. Bernardos, A. de la Oliva, P. Serrano, A. Banchs, L. M. Contreras, H. Jin, and J. C. Zuniga. “An architecture for software defined wireless networking”. In: *IEEE Wireless Communications* 21.3 (June 2014), pp. 52–61 (cit. on p. 82).

[6] A. Checko, H. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. Berger, and L. Dittmann. “Cloud RAN for Mobile Networks - a Technology Overview”. In: *I E E E Communications Surveys and Tutorials* 17.1 (2014), pp. 405–426. ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2355255 (cit. on pp. 80, 87).

[7] J. Costa-Requena, R. Kantola, J. Llorente, V. Ferrer, J. Manner, A. Y. Ding, Y. Liu, and S. Tarkoma. “Software defined 5G mobile backhaul”. In: *5G for Ubiquitous Connectivity (5GU), 2014 1st International Conference on*. Nov. 2014, pp. 258–263 (cit. on pp. 83, 86).

[8] J. Costa-Requena. “SDN integration in LTE mobile backhaul networks”. In: *The International Conference on Information Networking 2014 (ICOIN2014)*. Phuket, Thailand, Feb. 2014, pp. 264–269 (cit. on p. 82).

[9] A. Csoma, B. Sonkoly, L. Csikor, F. Németh, A. Gulyás, D. Jocha, J. Elek, W. Tavernier, and S. Sahhaf. “Multi-layered Service Orchestration in a Multi-domain Network Environment”. In: *2014 Third European Workshop on Software Defined Networks*. Sept. 2014, pp. 141–142 (cit. on p. 95).

[10] M. Dräxler and H. Karl. “Dynamic Backhaul Network Configuration in SDN-based Cloud RANs”. In: *CoRR* abs/1503.03309 (2015). URL: http://arxiv.org/abs/1503.03309 (cit. on p. 83).

[11] M. L. (Editor), A. G. (Editor), and M. Y. (Editor). *Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture*. Wiley, Aug. 2015 (cit. on p. 79).

[12] F. M. Facca, E. Salvadori, H. Karl, D. R. López, P. A. A. Gutiérrez, D. Kostic, and R. Riggio. “NetIDE: First Steps towards an Integrated Development Environment for Portable Network Apps”. In: *2013 Second European Workshop on Software Defined Networks*. Oct. 2013, pp. 105–110 (cit. on p. 98).

[13] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. “Participatory networking: an API for application control of SDNs”. In: *ACM SIGCOMM 2013*. Hong Kong, China, Aug. 2013, pp. 327–338 (cit. on p. 97).

[14] M. Fiorani, A. Rostami, L. Wosinska, and P. Monti. “Transport Abstraction Models for an SDN-Controlled Centralized RAN”. In: *IEEE Communications Letters* 19.8 (Aug. 2015), pp. 1406–1409 (cit. on p. 84).

[15] *Floodlight official web site*. http://www.projectfloodlight.org/floodlight/. Accessed: March 2016 (cit. on p. 90).

[16] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. “Languages for software-defined networks”. In: *IEEE Communications Magazine* 51.2 (Feb. 2013), pp. 128–134 (cit. on p. 98).

[17] A. Gudipati, D. Perry, L. E. Li, and S. Katti. “SoftRAN: Software Defined Radio Access Network”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China, 2013, pp. 25–30. ISBN: 978-1-4503-2178-5 (cit. on p. 83).

[18] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. “Practical Declarative Network Management”. In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN ’09. Barcelona, Spain: ACM, 2009, pp. 1–10 (cit. on p. 98).

[19] *Iperf official web site*. http://software.es.net/iperf/. Accessed: March 2016 (cit. on p. 91).

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. “B4: Experience with a globally-deployed software defined WAN”. In: *ACM SIGCOMM 2013*. Hong Kong, China, Aug. 2013, pp. 3–14 (cit. on p. 81).

[21] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. “SoftCell: Scalable and Flexible Cellular Core Network Architecture”. In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’13. Santa Barbara, California, USA, 2013, pp. 163–174. ISBN: 978-1-4503-2101-3 (cit. on pp. 82, 86).

[22] H. Kim and N. Feamster. “Improving network management with software defined networking”. In: *IEEE Communications Magazine* 51.2 (Feb. 2013), pp. 114–119 (cit. on p. 98).

[23] B. Lantz, B. Heller, and N. McKeown. “A network in a laptop: rapid prototyping for software-defined networks”. In: *Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Monterey, CA, US, Oct. 2010 (cit. on p. 90).

[24] M. Bjorklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020. IETF, Oct. 2010. URL: https://tools.ietf.org/html/rfc6020 (cit. on p. 101).

[25] E. Metsala and J. Salmelin. *Mobile Backhaul*. Wiley, Mar. 2012 (cit. on p. 84).

[26] Moore, B. and Ellesson, E. and Strassner, J. and Westerinen, A. *Policy Core Information Model – Version 1 Specification*. RFC 3060. IETF, Feb. 2001 (cit. on pp. 97, 99).

[27] *ONOS Intent Framework*. https://wiki.onosproject.org/display/ONOS/Intent+Framework. Accessed: March 2016 (cit. on p. 97).

[28] *OpenDaylight Application Policy*. https://wiki.opendaylight.org/view/Project_Proposals:Application_Policy_Plugin. Accessed: March 2016 (cit. on p. 97).

[29] *OpenDaylight NEMO*. https://wiki.opendaylight.org/view/NEMO:Main. Accessed: March 2016 (cit. on pp. 95, 97, 98).

[30] *OpenDaylight Network Intent Composition*. https://wiki.opendaylight.org/view/Project_Proposals:Network_Intent_Composition. Accessed: March 2016 (cit. on pp. 95, 97).

[31] *OpenDaylight official web site*. http://www.opendaylight.org/. Accessed: March 2016 (cit. on pp. 98, 101).

[32] *Openstack Congress*. https://wiki.openstack.org/wiki/Congress. Accessed: March 2016 (cit. on pp. 95, 97, 99–101).

[33] *Openstack Group Based Policy*. https://wiki.openstack.org/wiki/GroupBasedPolicy. Accessed: March 2016 (cit. on pp. 97, 98).

[34] *Openstack Heat*. https://wiki.openstack.org/wiki/Heat/AutoScaling. Accessed: March 2016 (cit. on pp. 95, 98).

[35] *Openstack Neutron QoS*. https://wiki.openstack.org/wiki/Neutron/QoS. Accessed: March 2016 (cit. on pp. 95, 98).

[36] S. Peng, B. Guo, C. Jackson, R. Nejabati, F. Agraz, S. Spadaro, G. Bernini, N. Ciulli, and D. Simeonidou. "Multi-Tenant Software-Defined Hybrid Optical Switched Data Centre". In: *Journal of Lightwave Technology* 33.15 (Aug. 2015), pp. 3224–3233 (cit. on p. 95).

[37] K. Pentikousis, Y. Wang, and W. Hu. "Mobileflow: Toward software-defined mobile networks". In: *IEEE Communications Magazine* 51.7 (July 2013), pp. 44–53 (cit. on p. 82).

[38] A. Pizzinat, P. Chanclou, T. Diallo, and F. Saliou. "Things you should know about fronthaul". In: *2014 The European Conference on Optical Communication (ECOC)*. Sept. 2014, pp. 1–3 (cit. on p. 80).

[39] D. Raymer, J. Strassner, E. Lehtihet, and S. van der Meer. "End-to-end model driven policy based network management". In: *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*. June 2006, (cit. on p. 97).

[40] P. Rost, C. J. Bernardos, A. D. Domenico, M. D. Girolamo, M. Lalam, A. Maeder, D. Sabella, and D. Wübben. "Cloud technologies for flexible 5G radio access networks". In: *IEEE Communications Magazine* 52.5 (May 2014), pp. 68–76 (cit. on p. 83).

[41] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. IETF, Mar. 2015 (cit. on p. 103).

[42] D. C. Verma. "Simplifying network administration using policy-based management". In: *IEEE Network* 16.2 (Mar. 2002), pp. 20–26 (cit. on p. 95).

[43] Y. Snir and Y. Ramberg and J. Strassner and R. Cohen and B. Moore. *Policy Quality of Service (QoS) Information Model*. RFC 3644. IETF, Nov. 2003 (cit. on p. 97).

[44] Y. Xia and S. Jiang and T. Zhou and S. Hares and Y. Zhang. *NEMO (NEtwork MOdeling) Language*. RFC draft-xia-sdnrg-nemo-language-04. IETF, Apr. 2016 (cit. on p. 98).

References from Chapter 7

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data Center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India, 2010, pp. 63–74. ISBN: 978-1-4503-0201-2 (cit. on p. 119).

[2] T. Benson, A. Akella, and D. A. Maltz. “Network Traffic Characteristics of Data Centers in the Wild”. In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC ’10. Melbourne, Australia, 2010, pp. 267–280. ISBN: 978-1-4503-0483-2 (cit. on p. 119).

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. “Hedera: Dynamic Flow Scheduling for Data Center Networks”. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI’10. San Jose, California, 2010, pp. 19–19 (cit. on p. 119).

[4] N. Farrington and A. Andreyev. “Facebook’s data center network architecture”. In: *2013 Optical Interconnects Conference*. May 2013, pp. 49–50 (cit. on p. 118).

[5] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. “Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers”. In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM ’10. New Delhi, India: ACM, 2010, pp. 339–350 (cit. on pp. 110, 111).

[6] A. Hammadi and L. Mhamdi. “A survey on architectures and energy efficiency in Data Center Networks”. In: *Computer Communications* 40 (2014), pp. 1–21 (cit. on p. 109).

[7] *Iperf official web site*. `http://software.es.net/iperf/`. Accessed: March 2016 (cit. on p. 118).

[8] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. “The Nature of Data Center Traffic: Measurements & Analysis”. In: *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*. IMC ’09. Chicago, Illinois, USA, 2009, pp. 202–208. ISBN: 978-1-60558-771-4 (cit. on p. 119).

[9] J. Kim, W. J. Dally, and D. Abts. “Flattened Butterfly: A Cost-efficient Topology for High-radix Networks”. In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 126–137 (cit. on p. 118).

[10] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter. “Circuit Switching Under the Radar with REACToR”. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI’14. Seattle, WA, 2014, pp. 1–15. ISBN: 978-1-931971-09-6 (cit. on pp. 111, 119).

[11] ONF. *OpenFlow Switch Specification, version 1.3.0.* Standard. Open Networking Foundation, 2012 (cit. on p. 112).

[12] P. Phaal and S. Panchen and N. McKee. *InMon Corporation’s sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. RFC 3176. IETF, Sept. 2001 (cit. on p. 113).

[13] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *SIGCOMM Comput. Commun. Rev.* 45.4 (Aug. 2015), pp. 183–197. ISSN: 0146-4833 (cit. on p. 118).

[14] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. "c-Through: part-time optics in data centers". In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2010). ISSN: 0146-4833 (cit. on pp. 110, 111).

References from Chapter 8

[1] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. "B4: Experience with a globally-deployed software defined WAN". In: *ACM SIGCOMM 2013*. Hong Kong, China, Aug. 2013, pp. 3–14 (cit. on p. 124).